문제해결을 위한

창의적 알고리즘(중급)

■ 집필진

안성진(성균관대학교 교수)
송태옥(가톨릭관동대학교 교수)
장승연(성균관대학교 연구원)
정종광(경기과학고등학교 교사)
배준호(경남정보고등학교 교사)
김봉석(경남과학고등학교 교사)
오은희(창원과학고등학교 교사)
정혜진(경기과학고등학교 교사)
전현석(경기과학고등학교 교사)
문광식(세종특별자치교육청 교사)
장원영(충북교육정보원 교사)
최웅선(수원정보과학고등학교 교사)
이 진(인천과학고등학교 교사)
박병기(서울과학고등학교 교사)
김종혜(경기과학고등학교 교사)
한건우(경기모바일과학고등학교 교사)
정웅열(경기북과학고등학교 교사)
배성환(광주과학고등학교 교사)
박소영(부산일과학고등학교 교사)

▌목 차▐

Part

I

문제해결과 알고리즘을 위한 준비 단계

1. 무료 C언어 통합개발환경 기본 사용 방법
2. 정보과학과 문제
3. 알고리즘과 실행시간측정

Part I

문제해결과 알고리즘을 위한 준비 단계

이 교재는 C/C++언어를 1년 이상 다룬 학생으로 기본적인 프로그래밍 언어의 문법은 모두 아는 학생을 대상으로 한다. 따라서 기본적인 프로그래밍 언어의 문법에 대한 설명은 하지 않으며, 알고리즘 설계를 위주로 구성된 교재이다.

이 단원에서는 무료로 이용할 수 있는 C/C++언어 통합개발환경의 설치 및 기본적인 사용법에 대해서 먼저 소개하고, 정보과학에서 다루는 문제의 특성 및 이를 해결하기 위하여 설계된 알고리즘의 성능을 분석하는 방법에 대해서 다룬다.

1 무료 C언어 통합개발환경 기본 사용 방법

프로그래밍 언어를 활용하는 문제 해결을 위해서는 자신이 생각한 문제 해결 방법과 알고리즘을 프로그래밍 언어로 작성하고 실행한 후, 출력되는 결과를 자신이 예상한 결과와 비교해 보는 과정이 필요하다.

가장 많이 사용되는 프로그래밍 언어 중 하나인 C/C++ 코드를 작성하고, 실행 결과를 확인하고, 실행파일로 만들 수 있는 많은 통합개발환경(IDE)이 있지만, 상업적 판매 목적의 소프트웨어를 만들어내기 위한 큰 프로젝트가 아닌 이상 고가의 유료 통합개발환경을 구입해서 사용할 필요는 없다.

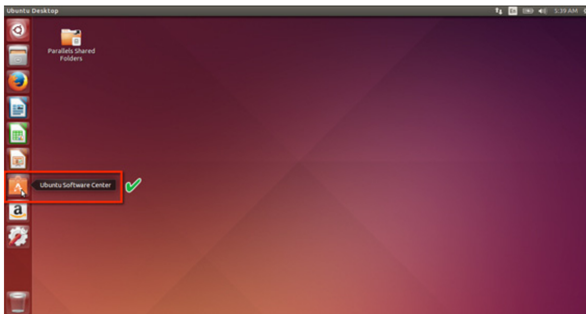
C/C++ 언어로 작성된 코드들은 일반적으로, 국제적으로 제정되어있는 표준 방법들을 사용하면 모든 통합개발환경에서 정상적으로 컴파일 되고 동작하기 때문이다.

다양한 프로그램들이 있지만, 무료로 사용할 수 있으면서도 편리하고 강력한 몇 가지 C/C++ 통합개발환경에 대해서 살펴보자.

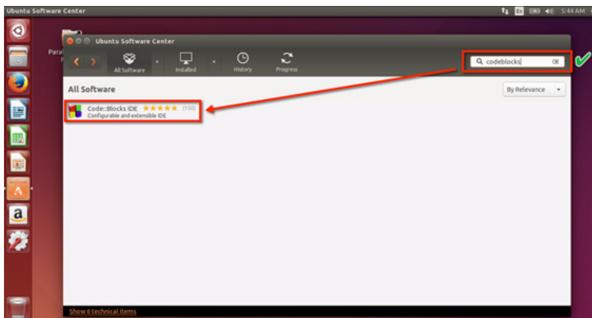
가. Ubuntu Linux && Code::Blocks

코드블락스(Code::Blocks)는 매우 강력하면서도 지속적으로 버전업되고 있는 무료 공개 IDE로, Windows/Linux/MacOSX 운영체제에서 모두 사용할 수 있다.

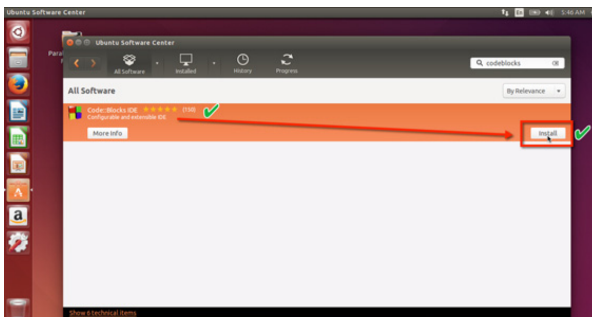
Ubuntu Linux에 Code::Blocks를 설치하고 간단히 실행하는 방법은 다음과 같다.
(공식사이트 : http://www.codeblocks.org/에서는 운영체제에 따른 설치 파일을 제공하고 있다.)



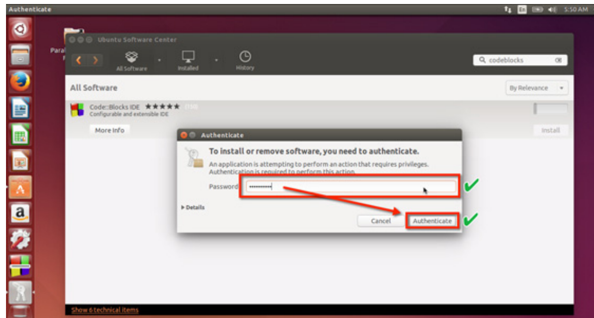
우분투 운영체제에서 왼쪽 메뉴의 우분투 소프트웨어 센터(Ubuntu Software Center)를 클릭한다.



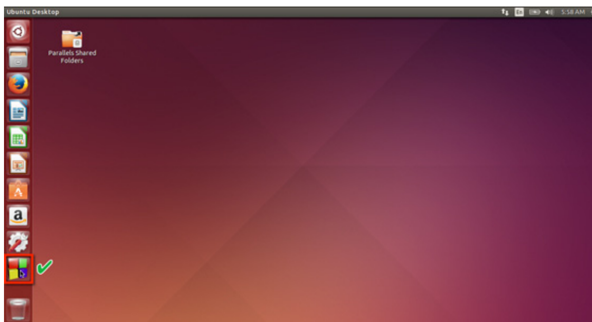
검색창에서 codeblocks를 검색한다.



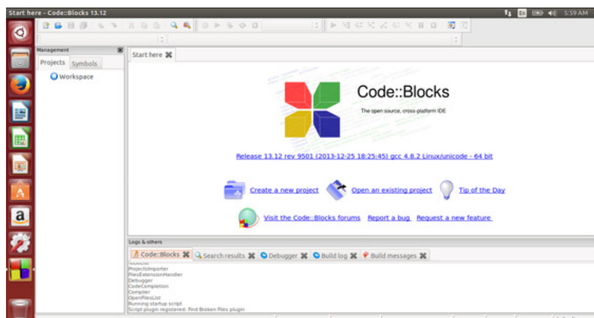
codeblocks를 누른 후 활성화되는 install을 누른다.



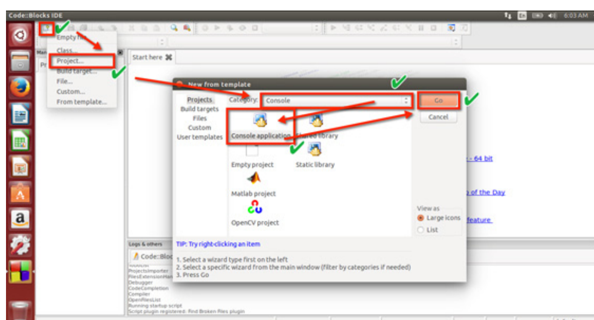
설치를 위해 비밀번호를 입력하고 설치를 진행한다.



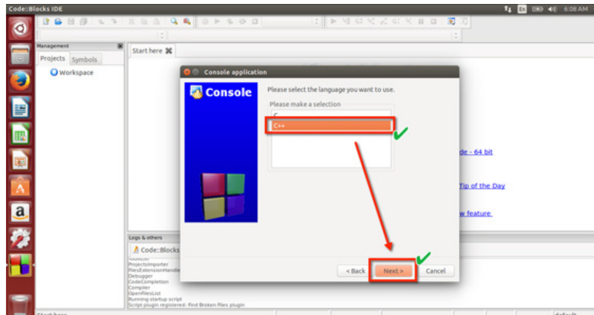
설치가 완료되면 왼쪽 메뉴에서 코드블락스 아이콘을 눌러 실행시킨다.



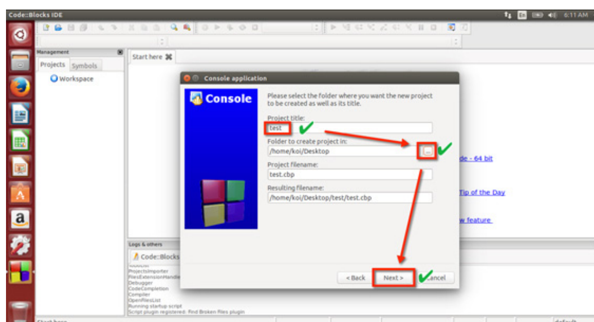
코드블락스 실행을 확인한다.



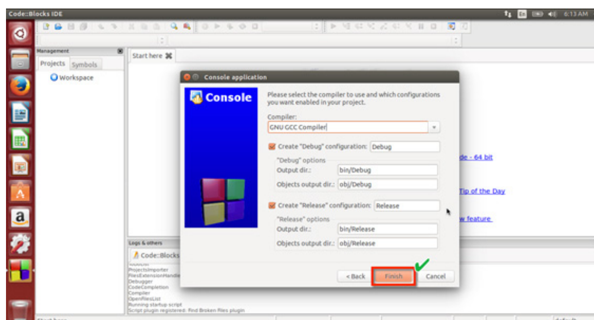
프로젝트(폴더)를 만들기 위해 New-Project를 눌러 실행하고, Category에서 Console를 선택한다. 아래에서 Console application을 선택하고 GO를 누른다.



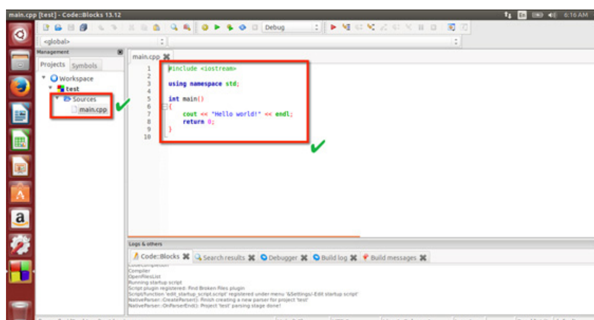
작성할 언어 종류를 C++로 선택하고 Next를 누른다.



프로젝트 폴더명, 저장할 위치(Desktop은 처음 바탕화면을 의미)를 선택하고 Next를 누른다.



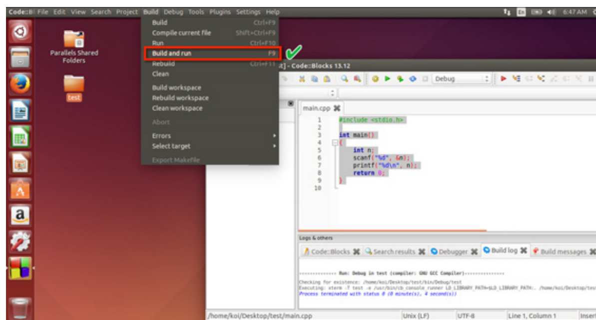
Finish를 누르면 표준 GCC 컴파일러로 설정이 되고, 코드 작성 준비가 끝난다.



왼쪽 탐색 창에서 Sources 폴더에서 main.cpp 파일을 눌러보면 기본 C++ 코드가 들어있고 편집할 수 있게 된다.

아래와 같은 기본 코드를 작성해 넣어보자.

코드	참고
#include <stdio.h>	
int n;	
int main()	
{	
scanf("%d", &n);	
printf("%d\n", n);	
return 0;	
}	

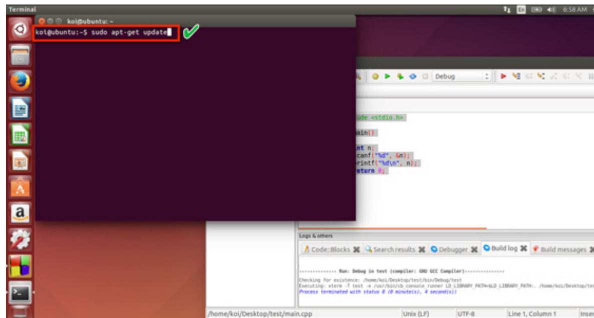


작성한 코드를 컴파일하고 실행하기 위해서는 코드블럭스 메뉴에서 Build – Build and run을 누르면 된다.

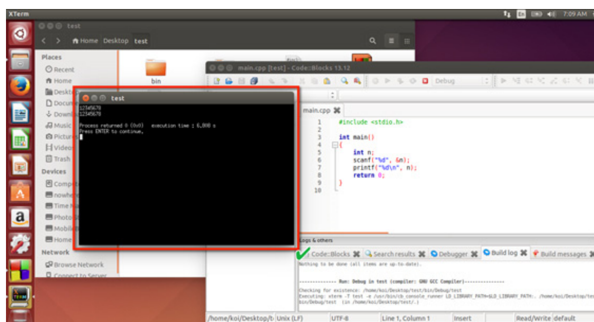
처음 설치하였을 때에는 g++ not found 와 같은 컴파일 오류가 발생할 수 있다. 그럴 때에는 터미널 명령 입력 창을 열고 몇 가지 명령을 실행시키면 해결된다.



Dash home을 실행하고 검색창에서 terminal을 찾아 실행시킨 후



명령 입력 창에서
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install build-essential
gcc –v
make –v
를 순서대로 실행하면 코드블락스에서 작성한 소스코드를 컴파일/실행 가능



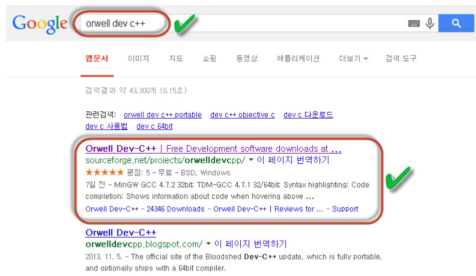
Build – Build and run을 누르면

실행창으로 실행이 되면서 입출력이 가능하게 된다.

나. Windows && Orwell DevC++

Orwell DevC++는 무료 공개 IDE인데 Windows 운영체제에서만 사용할 수 있다. 한국 KLDP 그룹에 의해 한글화가 제공되어, 한글 메뉴를 사용할 수 있고, 기본 설치만으로도 한글이 포함된 소스코드를 사용할 수 있다.

Windows에 Orwell DevC++를 설치하고 간단히 실행하는 방법은 다음과 같다(공식 사이트 : http://orwelldevcpp.blogspot.kr/에서는 몇 가지 다른 설치 파일을 제공하고 있는데, 일반 버전은 http://sourceforge.net/projects/orwelldevcpp/를 통해 다운로드하고 설치할 수 있다.).

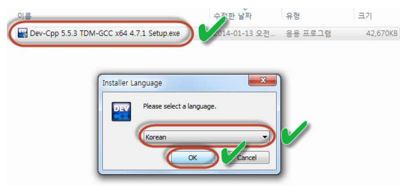


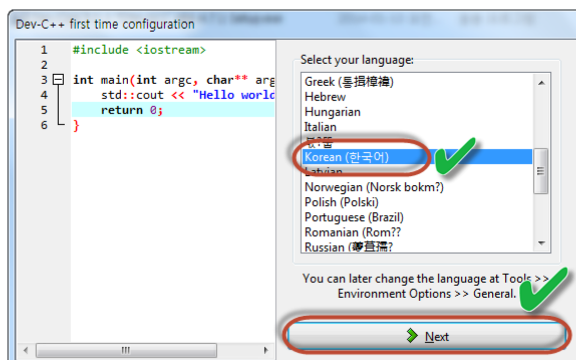
orwell dev c++를 검색하고 다운로드 사이트를 누른다.



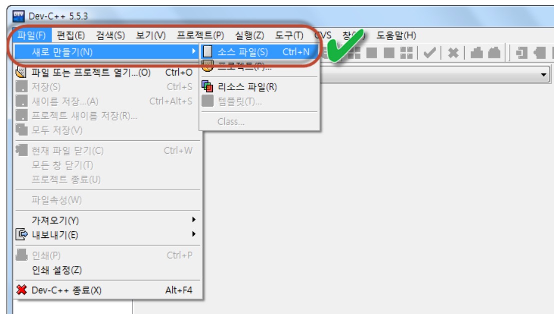
공식 배포 사이트를 통해 설치 파일 다운로드 링크를 누른다.

설치 언어로 한국어 설정



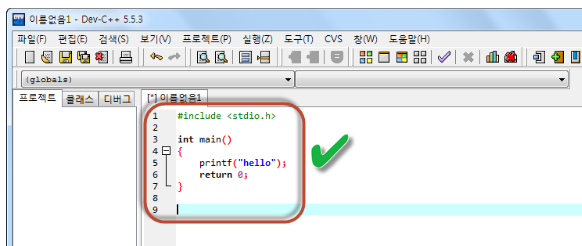


기본 언어로 한국어를 선택하여 설치 완료



프로그램을 실행시킨 후

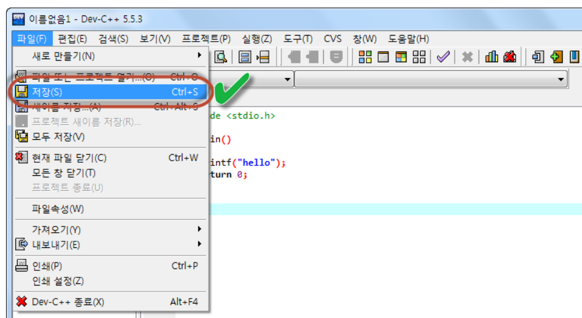
파일 – 새로 만들기 – 소스 파일을 눌러 코드를 작성할 수 있다.
(프로젝트를 만들지 않아도 소스 파일만 작성하고도 실행이 가능하다.)



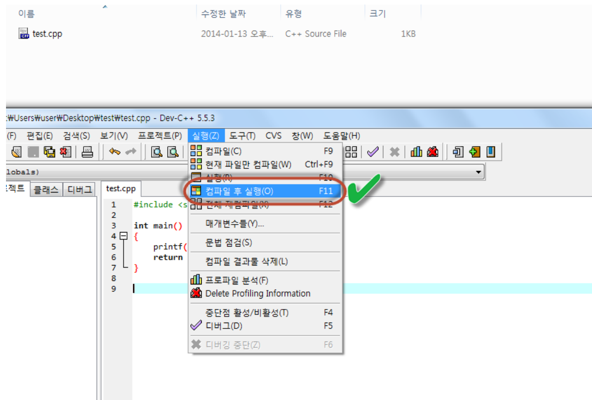
소스 코드를 작성한다.

아래와 같은 기본 코드를 작성해 넣어보자.

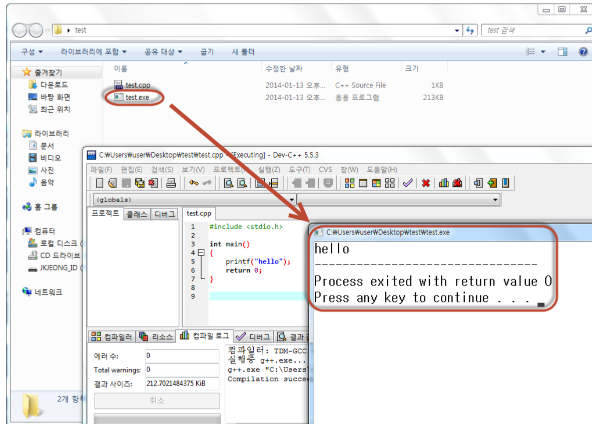
줄	코드	참고
1	#include <stdio.h>	
2		
3	int main()	
4	{	
5	printf("hello");	
6	return 0;	
7	}	



소스 코드를 파일로 저장한다.



실행 – 컴파일 후 실행을 누르면 실행 파일이 만들어지고 실행 결과를 확인할 수 있다.



실행 결과 확인

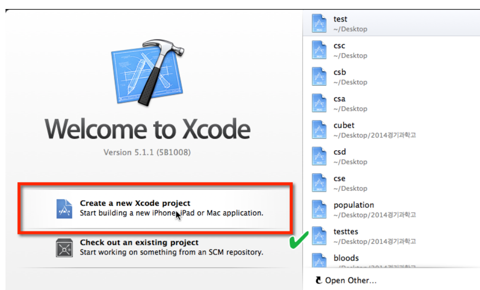
다. Mac OS X && Xcode

Xcode는 무료 IDE인데 Mac OS X 운영체제에서만 사용할 수 있다. Xcode는 간단한 C/C++ 프로그래밍부터 OS X용 응용프로그램 개발, iOS용 앱 개발이 가능한 전문 IDE이다.

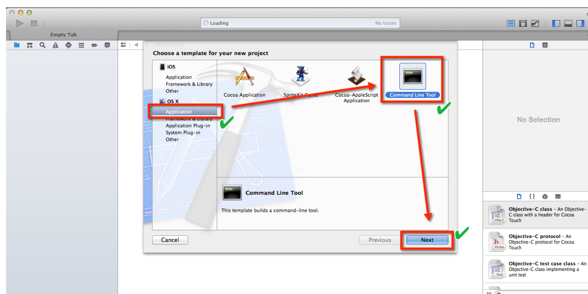
Mac OS X 에 Xcode를 설치하고 간단히 실행하는 방법은 다음과 같다(Mac OS X 의 앱스토어를 이용해 다운로드하고 설치한다.).



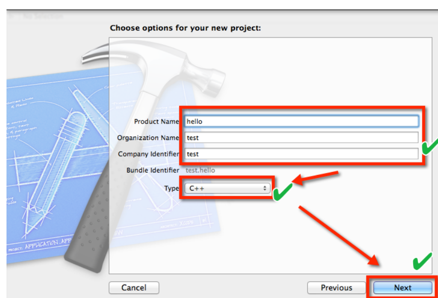
앱스토어를 통해 Xcode를 검색해 설치한다.



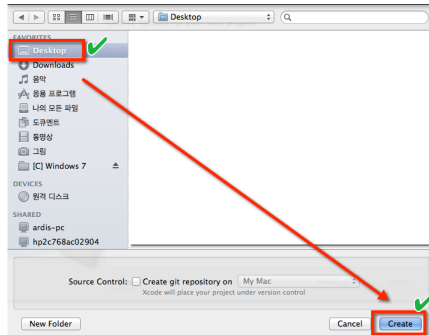
설치된 Xcode 실행 후
Create a new Xcode project
를 누른다.



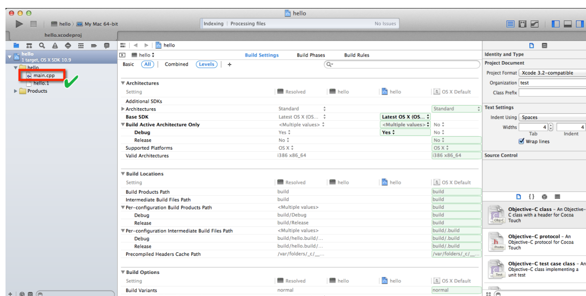
OS X – Application을 선택하고
Command Line Tool을 선택한다.
Next를 누른다.



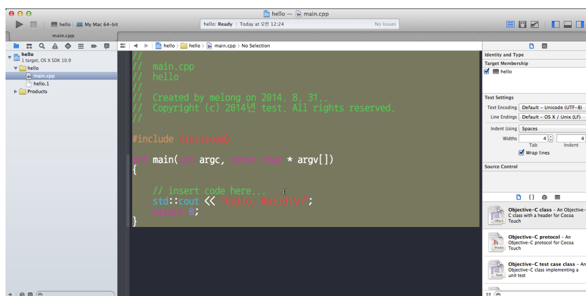
프로젝트 폴더 이름 등을 입력한다.
작성할 언어 타입을 C++ 로 지정한다.
Next를 누른다.



프로젝트 폴더를 만들 위치를 선택한다.
(Desktop은 바탕화면)



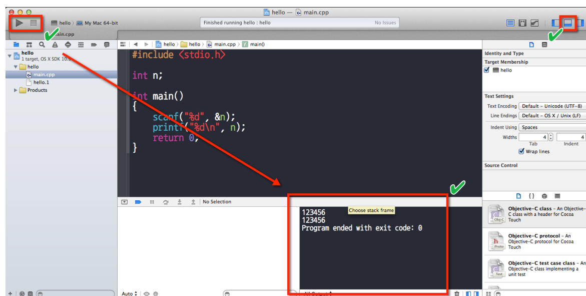
왼쪽 탐색창에서 main.cpp를 누른다.



소스 코드를 작성한다.

아래와 같은 기본 코드를 작성해 넣어보자

줄	코드(	참고
1	#include <stdio.h>	
2		
3	int n;	
4		
5	int main()	
6	{	
7	scanf("%d", &n);	
8	printf("%d\n", n);	
9	return 0;	
10	}	



하단 창을 보이게 하고(오른쪽 위 아이콘에서 선택)

실행/정지 버튼을 누른다(왼쪽 위 아이콘에서 선택).

입력 출력이 하단 창에서 이루어진다.

라. 파일입출력 기본

채점을 위한 입출력이 파일로 설정되는 경우에는 지정된 파일로 데이터를 읽어 들이거나, 결과를 작성해야 한다. 아래는 가장 기본적인 입출력 예시이다.

- 표준 입출력

줄	코드	참고
1	#include <stdio.h>	7: 형식으로 입력 8: 형식에 맞추어 출력
2		
3	int n;	
4		
5	int main()	
6	{	
7	scanf("%d", &n);	
8	printf("%d\n", n);	
9	return 0;	
10	}	

– freopen( ) 표준입출력 재할당(변환)

줄	코드	참고
1	`#include <stdio.h>`	7: 표준입력(콘솔입력)의 내용을 in.txt 파일에서 읽어오도록 재할당 8: 표준출력(콘솔출력)의 내용을 out.txt 파일에 기록하도록 재할당 9: 형식으로 입력 10: 형식에 맞추어 출력
2		
3	`int n;`	
4		
5	`int main()`	
6	`{`	
7	`  freopen("in.txt","r",stdin);`	
8	`  freopen("out.txt","w",stdout);`	
9	`  scanf("%d", &n);`	
10	`  printf("%d\n", n);`	
11	`  return 0;`	
12	`}`	

– fopen( ) 파일처리

줄	코드	참고
1	`#include <stdio.h>`	7: in.txt 파일에서 데이터를 읽어오도록 지정 8: out.txt 파일로 데이터를 출력하도록 지정 9: 파일에서 형식으로 입력 10: 파일로 형식에 맞추어 출력
2		
3	`int n;`	
4		
5	`int main()`	
6	`{`	
7	`  FILE *in=fopen("in.txt","r");`	
8	`  FILE *out=fopen("out.txt","w");`	
9	`  fscanf(in,"%d", &n);`	
10	`  fprintf(out,"%d\n", n);`	
11	`  return 0;`	
12	`}`	

실행파일 생성이나 파일 입출력은 운영체제의 사용자 권한과 관련한 문제들이 발생할 수 있기 때문에 때에 따라서 소스코드를 작성하고 실행시키기 위해서 추가적인 설정 작업이 필요한 경우도 있다.

2 정보과학과 문제

이 교재에서는 지금까지 수학에서 다루어 온 문제들을 조금 다른 관점으로 분류하고 정의한다. 이 교재에서는 문제를 단순계산 문제, 결정 문제, 최적화 문제로 나누어 다룬다.

가. 계산 문제

정보과학에서 계산 문제(computational problem)는 수학적으로 계산 가능하며, 컴퓨터를 이용하여 풀 수 있는 모든 문제들을 의미한다.

예를 들어, "자연수 n이 주어질 때, n의 약수를 모두 구하시오." 와 같은 문제는 계산 문제이다. 이와 같이 계산 문제란 잘 정해진 규칙으로 인해 아무런 오해의 소지가 없이 정해진 방법으로 규칙을 적용하면 답을 구할 수 있는 모든 문제이다. 쉽게 말하자면 계산 가능한 문제를 의미한다. 여기서 말하는 계산이란 단순히 사칙연산의 수준을 벗어난다. 다음 예를 통하여 계산 문제와 그렇지 않은 문제를 이해하자.

[표-1] 계산이 가능한 문제와 계산이 불가능한 문제

계산이 가능한 문제	계산이 불가능한 문제
① N 이하의 자연 수 중 짝수의 개수는? ② 폐구간 $[a,\ b]$에서 소수는 존재하는가? ③ n개의 원소를 가지는 집합 S의 원소를 오름차순으로 나열하고, 그 최댓값을 구하시오. ④ N개의 원소로 이루어진 집합의 부분집합들 중 그 원소의 합이 k인 부분집합이 존재하는가? ⑤ 좌표평면 상에 N개의 점의 한 점에서 출발하여 모든 점을 지나고 출발점으로 돌아오는 경로의 길이 중 가장 짧은 길이는 얼마인가?	① 우리 반 학생들 중 키가 큰 학생은 몇 명인가? ② 주어진 음식들 중 맛있는 순서로 나열하시오. ③ 대한민국에서 축구를 잘하는 사람은 모두 몇 명인가? ④ 임의로 한 명을 골랐을 때, 그 사람이 요리를 잘 할 확률은 얼마인가?

일반적으로 정보과학에서는 계산 가능한 문제에 대해 결정 문제(decision problem), 탐색 문제(search problem), 카운팅 문제(couting problem), 최적화 문제(optimization problem),

함수형 문제(function problem) 등으로 나누지만, 이 책에서는 크게 "결정 문제"와 "최적화 문제"로 구분하여 다룬다.

일반적으로 정보올림피아드를 비롯한 각종 프로그래밍 대회에서는 최적화 문제나 NP－complete 문제[1]에서 입력값 n의 크기를 줄이거나, 제약조건을 두어 다루는 경우가 많다.

나. 결정 문제

결정 문제란 계산 문제들 중 그 결과를 'Yes', 'No' 중 하나로 답할 수 있는 문제를 의미한다. [표-1]에서 계산문제의 예시들 중 ②, ④와 같은 문제가 결정 문제이다.

결정 문제는 각종 프로그래밍 대회에서 직접적으로 출제되는 경우는 흔하지 않지만, 난이도가 높은 최적화 문제를 결정 문제의 형태로 바꾸어 해결하고, 그 결과들을 이용하여 해를 구하는 방법이 있다. 따라서 결정 문제를 간접적으로 다룰 줄 알아야 한다.

다. 최적화 문제

최적화 문제는 계산결과 얻은 후보 해들 중 가장 적절한 해를 찾는 형태의 문제를 말한다. [표-1]에서 계산 문제의 예시들 중 ③, ⑤와 같이 최댓값이나 가장 짧은 경로의 길이를 구하는 형태의 문제로 정보올림피아드를 비롯한 각종 프로그래밍 대회에서 가장 자주 출제되는 형태의 문제이다. 따라서 이 책에서는 최적화 문제를 많이 다룬다.

1) 결정 문제의 해를 검증하는 방법은 다항시간으로 가능하나, 해를 구하는 다항시간의 방법이 아직 알려지지 않은 결정 문제들의 집합으로, 3－SAT이라고 불리는 문제를 풀면 모두 풀 수 있음이 알려져 있다.

3 알고리즘과 실행시간 측정

가. 알고리즘

알고리즘의 정보과학적 정의에 대해서는 자세히 다루지 않겠지만 간단히 말하자면, 알고리즘이란 주어진 문제를 해결하기 위한 단계 혹은 절차를 말한다. 그리고 이 절차에는 입력값과 출력값이 존재해야하며, 유한한 단계를 거쳐서 반드시 종료되어야 한다.

일반적으로 프로그래밍 대회들에서는 이 유한한 단계를 제한시간으로 설정하여 주로 수 초 이내에 작성한 알고리즘이 해를 구할 수 있는지 여부로 유한성을 판단한다.

알고리즘은 주로 자연어, 의사코드, 프로그래밍언어 등의 방법으로 기술할 수 있다. 다음은 집합 S의 원소들의 합을 구하는 알고리즘 A를 각 방법으로 기술한 예이다.

[자연어]

알고리즘 A
(1단계) 원소의 인덱스를 id로 정의한다.
(2단계) 집합 S에 대하여 $\sum_{id=1}^{n} S_{id}$ 를 구하고 이를 s라 한다.
(3단계) s를 출력하고 종료한다.

[의사코드]

알고리즘 A
(1단계) $id \leftarrow 1$, $s \leftarrow 0$
(2단계) $s = s + S_{id}$, $id \leftarrow id+1$
(3단계) $id \le n$ $goto$ 2단계
(4단계) $print\ s$

[프로그래밍 언어 (C++)]

```
void A(int S[], int n)
{
   int s = 0;
   for(int id=1; id<=n; id++)
      s = s + S[id];
   printf("%d₩n",s);
}
```

이 책에서는 모든 알고리즘을 C++언어로 표현한다.

기본적으로 알고리즘에서 제시된 방법에 따라서 효율성이 달라진다. 알고리즘의 효율성을 측정하기 위하여 다양한 방법이 있지만 이 교재에서는 알고리즘의 효율성을 계산량으로 표현하며, 계산량은 입력크기 n에 대한 실행시간을 나타낸다.

위에서 제시된 알고리즘의 계산량은 입력크기 n이 커지면 실행시간은 n에 1차 함수적으로 비례하여 커진다는 것을 쉽게 알 수 있다. 따라서 위 알고리즘A의 계산량은 n이다. 그리고 이를 정보과학에서는 $O(n)$이라고 표현하기도 한다. 이와 같이 계산량을 나타내는 점근적 표현 방법으로 O, Θ, Ω 등이 있으나 이 교재에서는 따로 다루진 않는다.

나. 실행시간의 측정

정보올림피아드와 같은 각종 프로그래밍 경시대회나 세계적으로 운영되고 있는 Online Judge 등에서는 알고리즘의 성능을 주어진 입력에 대한 실행시간으로 측정한다.

일반적으로 실행시간은 CPU time만을 측정해야 하지만 정보올림피아드에서는 IO time까지 포함되기 때문에 입출력을 빠르게 작성하는 것도 어느 정도 도움이 된다.

실행시간을 측정하는 방법을 알아두면 알고리즘의 효율을 실험적으로 확인해 볼 수 있는데, 주어진 n개의 데이터를 정렬하는 문제를 이용해 실행 시간을 측정하는 방법에 대해서 알아본다.

먼저 무작위 정수 n개를 발생시키는 방법은 다음과 같다.

줄	코드	참고
1	`#include <stdio.h>`	5: 이 문제에서 입력값 n의 정의역은 3 <= n <= 100,000으로 한다.
2	`#include <stdlib.h>`	
3	`#include <time.h>`	
4		
5	`int n, S[100000];`	9: 랜덤 시드 값을 시간으로 결정
6		
7	`int main()`	12: rand()함수는 $0 \sim 2^k - 1$의 값을 발생(k의 값은 컴파일러에 따라 다르지만 보통 15, 31 중 하나임)
8	`{`	
9	`  srand(time(NULL));`	
10	`  scanf("%d", &n);`	
11	`  for(int i=0; i<n; i++)`	
12	`    S[i] = rand();`	
13	`  return 0;`	
14	`}`	

위 방법으로 작성하면 배열 S에 랜덤한 값 n개가 입력된다. 다음 줄에 출력문을 추가해 값을 확인해 보자.

줄	코드	참고
1	`#include <stdio.h>`	
2	`#include <stdlib.h>`	
3	`#include <time.h>`	
4	`int n, S[100000];`	
5	`int main()`	
6	`{`	
7	`  srand(time(NULL));`	
8	`  scanf("%d", &n);`	
9	`  for(int i=0; i<n; i++)`	
10	`    S[i] = rand();`	
11	`  for(int i=0; i<n; i++)`	
12	`    printf("%d ", S[i]);`	
13	`  return 0;`	
14	`}`	

[실행결과]

```
10
13426 30105 8303 21311 26182 6776 9981 29521 12570 26141
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

다음으로 정렬 알고리즘을 작성하여 n개의 자료를 정렬하는 시간을 측정한다. 먼저 선택정렬(selection sort)을 작성하여 측정한다.

줄 | 코드 | 참고

```
#include <stdio.h>
#include <stdlib.h>
#include <ctime>

int n, S[100000];

void print_array()
{
  for(int i=0; i<n; i++)
    printf("%d ", S[i]);
  printf("\n");
}

void swap(int a, int b)
{
  int t=S[a];
  S[a]=S[b];
  S[b]=t;
}

void selection_sort(void)
{
  for(int i=0; i<n-1; i++)
    for(int j=i+1 ; j<n; j++)
      if(S[i] > S[j])
        swap(i, j);
}

int main()
```

```
{
  srand(time(NULL));
  scanf("%d", &n);
  for(int i=0; i<n; i++)
    S[i] = rand();
//print_array();
  int start = clock();

  selection_sort();

  printf("result=%.3lf(sec)\n",(double)(clock()-start)/CLOCKS_PER_SEC);
//print_array();
  return 0;
}
```

빨간색 부분이 알고리즘의 실행시간을 측정하는 부분이다. 즉, selection_sort 함수의 실행시간을 측정하는 프로그램이다. print_array 출력함수는 n값이 커지면 출력결과가 너무 많기 때문에 주석처리를 한 것이다. 실행결과는 다음과 같다.

[실행결과]

[n=1,000]

```
1000
result=0.002(sec)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

[n=10,000]

```
10000
result=0.337(sec)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

[n=100,000]

```
100000
result=28.078(sec)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

보는 바와 같이 선택정렬은 자료가 10배 증가할수록 실행시간은 약 100배 증가함을 알 수 있다. 따라서 n배 커지면 실행시간은 n^2에 비례하여 증가한다.

퀵정렬(quick sort)의 경우는 선택정렬보다 훨씬 빠른 속도로 정렬할 수 있는 알고리즘이다. 여기서는 퀵정렬 기반으로 동작하는 std::sort( )를 이용해 실행시간을 측정한다.

줄	코드	참고
1	`#include <stdio.h>`	4: std::sort를 사용하기 위하여 추가
2	`#include <stdlib.h>`	
3	`#include <time.h>`	6: 100만개 까지 측정
4	`#include <algorithm>`	
5		
6	`int n, S[1000000];`	
7		
8	`void print_array()`	
9	`{`	
10	`  for(int i=0; i<n; i++)`	
11	`    printf("%d ", S[i]);`	
12	`  printf("\n");`	
13	`}`	
14		
15	`int main()`	
16	`{`	
17	`  srand(time(NULL));`	
18	`  scanf("%d", &n);`	
19	`  for(int i=0; i<n; i++)`	
20	`    S[i] = rand();`	
21	`//print_array();`	
22		
23	`  int start = clock();`	
24		
25	`  std::sort(S, S+n);`	

줄	코드	참고
	printf("result=%.3lf(sec)₩n", (double)(clock()-start)/CLOCKS_PER_SEC);	
	//print_array();	
	}	

[n=100,000]

```
100000
result=0.016(sec)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

[n=1,000,000]

```
1000000
result=0.187(sec)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

이와 같이 100만개까지 정렬 하는데 0.2초 이내로 처리할 수 있으므로, 알고리즘 효율의 차이를 직접 체감할 수 있다.

C++에서는 위 예제에서 사용한 std::sort() 와 같이 프로그래밍 대회에서 사용할 수 있는 다양한 함수를 제공한다. 이러한 함수의 사용법은 교재에서 사용할 때마다 소개하니, 사용법을 익혀두면 다양한 알고리즘에 응용할 수 있으므로 꼭 익혀두기 바란다.

std::sort()는 $O(n \lg n)$으로 자료를 정렬하는 함수이며, 배열, 구조체, std::vector, std::list, std::set, std::map 등의 다양한 형태의 자료구조를 모두 정렬할 수 있는 매우 강력한 정렬 함수이다. 기본적인 활용방법은 다음과 같다.

```
std::sort(정렬할 자료의 시작 주소, 정렬할 자료의 마지막 주소, [비교함수의 주소]);
```

using namespace std; 명령을 실행한 다음이라면 std::를 생략하고 사용할 수 있다. 비교함수는 일반적으로 compare라는 이름으로 만들며, 생략하면 오름차순으로 정렬한다. 만약 내림차순이거나 구조체 등의 정렬에서 우선순위가 필요할 때는 비교함수를 작성해야 한다. 다음은 비교함수의 작성법을 보여준다.

[오름차순의 경우]

```
bool compare(int a, int b) //정수 배열의 오름차순 정렬일 경우
{
  return a < b;              //왼쪽의 원소가 오른쪽의 원소보다 값이 작도록 정렬
}
```

[내림차순의 경우]

```
bool compare(int a, int b) //정수 배열의 내림차순 정렬일 경우
{
  return a > b;              //왼쪽의 원소가 오른쪽의 원소보다 값이 크도록 정렬
}
```

[x, y를 멤버로 하는 POINT 구조체에서 x를 기준으로 오름차순 정렬]

```
bool compare(POINT a, POINT b) //POINT 구조체 정렬
{
    return a.x < b.x;          //x멤버 기준으로 오름차순
}
```

[x, y를 멤버로 하는 POINT 구조체에서 1순위 x, 2순위 y 기준 오름차순 정렬]

```
bool compare(POINT a, POINT b)     // POINT 구조체 정렬
{
    if( a.x == b.x ) return a.y < b.y;
    else return a.x < b.x;         // x멤버 기준으로 오름차순
}
```

위의 4가지 예시를 잘 이해하면 다양한 형태로 활용할 수 있다. 위와 같이 compare함수를 정의하면 std::sort()를 다음과 같이 사용하면 된다.

[S 배열의 처음부터 n-1번째까지의 원소를 compare 함수의 정의대로 정렬]

```
std::sort(S, S+n, compare);
```

알고리즘 학습을 위해서는 std::sort()를 활용하는 것도 중요하지만 퀵정렬과 같은 알고리즘을 직접 구현해보는 연습도 반드시 필요하다.

실제 대회 중에는 주어진 시간 동안 정확한 알고리즘을 만들어내야 하므로, 일반적으로 STL(Standard Template Library)를 활용할 줄 아는 것도 중요하다. STL이란 앞에서 언급한 sort, list, vector, set, map과 같은 표준 템플릿 라이브러리를 말한다.

마지막으로 실제로 퀵정렬을 직접 작성해 실행시간을 측정해 보자. 퀵정렬 알고리즘의 큰 틀은 다음과 같다.

1. pivot(기준값)을 정한다.
2. pivot보다 작은 원소들은 왼쪽으로, pivot보다 큰 원소들은 오른쪽으로 보낸다.
3. pivot을 기준으로 왼쪽 배열과 오른쪽 배열을 새로운 배열로 정하고, 각 배열 구간에 대해서 1번 과정을 재귀적으로 반복한다.

퀵정렬을 구현하는 여러 가지 방법들이 있다. 특히 1번 과정에서처럼 pivot(기준값)을 정하는 여러 가지 아이디어들이 있으나, 일반적으로 처음의 원소 또는 가장 마지막 원소를 pivot으로 잡는 방법을 사용한다.

하지만 이렇게 pivot을 잡았을 때 최악의 경우, 즉 역순으로 정렬되어 있을 경우 계산량이 $O(n^2)$이 될 수 있다. 하지만 이런 경우는 $\frac{1}{n!}$의 확률로 거의 발생하지 않는다. 만약 발생한다 하더라도 처음에 한 번 검사하여 그냥 뒤집으면 되기 때문에 일반적으로 큰 문제가 되지는 않는다.

수학적으로 이러한 경우가 절대로 발생하지 않도록 pivot을 잡는 다양한 방법이 있지만 이 예제에서는 단순히 처음에 나오는 원소를 pivot으로 잡는 방법으로 구현한다. 소스코드는 다음과 같다.

```
void swap(int a, int b)
{
  int t = S[a];
  S[a] = S[b];
  S[b] = t;
}

void quick_sort(int s, int e)
{
  if(s<e)
  {
    int p = s, l = s+1, r = e;
    while(l<=r)
    {
      while(l<= e && S[l]<=S[p]) l++;
      while(r>=s+1 && S[r]>=S[p]) r--;
      if(l<r) swap(l,r);
    }
    swap(p, r);
    quick_sort(s, r-1);
    quick_sort(r+1, e);
  }
}
```

직접 작성한 퀵정렬의 시간을 측정해보자.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <algorithm>
5
6  int n, S[1000000];
7
8  void print_array()
9  {
10   for(int i=0; i<n; i++)
11     printf("%d ", S[i]);
12   printf("\n");
13 }
```

참고:

4: std::sort를 사용하기 위하여 추가

6: 100만 개 까지 측정

코드	참고

```

void swap(int a, int b)
{
  int t = S[a];
  S[a] = S[b];
  S[b] = t;
}

void quick_sort(int s, int e)
{
  if(s<e)
  {
    int p = s, l = s+1, r = e;
    while(l<=r)
    {
      while(l<= e && S[l]<=S[p]) l++;
      while(r>=s+1 && S[r]>=S[p]) r--;
      if(l<r) swap(l,r);
    }
    swap(p, r);
    quick_sort(s, r-1);
    quick_sort(r+1, e);
  }
}

int main()
{
  srand(time(NULL));
  scanf("%d", &n);
  for(int i=0; i<n; i++)
    S[i] = rand();
//print_array();

  int start = clock();

  quick_sort(0, n-1);

  printf("result=%.3lf(sec)\n",(double)(clock()-start)/CLOCKS_PER_SEC);

//print_array();
  return 0;
}
```

[n=100,000]

```
100000
result=0.016(sec)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

[n=1,000,000]

```
1000000
result=0.206(sec)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Part

II

탐색기반 알고리즘의 설계

Part II

탐색기반 알고리즘의 설계

이 교재에서는 알고리즘 설계 방법을 크게 탐색기반 설계방법과 관계기반 설계방법으로 나누어 다룬다. 이렇게 분류한 것은 학생들이 알고리즘을 보다 쉽고, 체계적으로 설계하기 위한 편의상의 분류이다.

탐색기반 설계방법은 문제를 탐색 가능한 형태로 구조화한 후 탐색해 나가는 알고리즘 설계방법이다.

하지만 대부분의 경우에는 탐색해야 할 공간이 너무 크기 때문에 문제에서 요구하는 시간 내에 해를 구하지 못하는 경우가 많다. 따라서 전체탐색법을 적절히 응용하는 알고리즘 설계법들을 익힐 필요가 있다.

이 단원에서는 전체탐색법을 기반으로 한 다양한 응용을 익히기 위하여 다양한 탐색방법을 먼저 학습하고, 학습한 탐색방법을 활용하여 주어진 문제를 해결할 수 있는 알고리즘 설계를 실습한다.

4 탐색

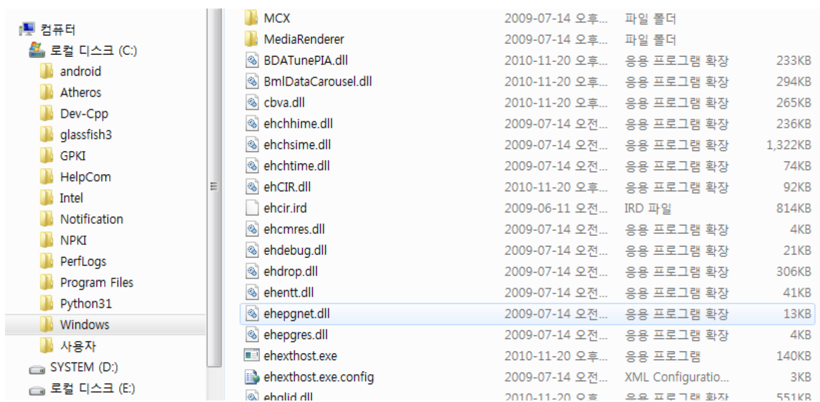
정보과학에서 탐색은 매우 중요하게 다루어지는 주제 중 하나이다. 일반적으로 탐색이란 같은 형태의 한 개 이상의 자료들이 적절한 형태로 구조화된 상태로 모여 있는 집합에서 특정 자료를 찾는 모든 작업을 말한다.

따라서 탐색은 탐색할 자료가 저장되어 있는 구조를 먼저 파악하는 것이 중요하다. 문제의 특성에 따라 구조가 명시적으로 드러나는 경우는 쉬운 문제에 속하고, 문제에서는 탐색 구조가 직접적으로 드러나지는 않으나, 문제를 해결하는 과정에서 자체적으로 구조화하며 탐색하는 경우는 중급 이상의 문제라 할 수 있다.

이 교재에서는 탐색의 대상이 되는 구조를 선형구조와 비선형구조로 나누어 탐색을 실습한다. 일반적으로 배열이나 연결리스트로 표현될 수 있는 구조를 선형구조, 트리나 그래프의 형태로 표현되는 구조를 비선형구조로 구분한다.

탐색의 대상이 되는 구조화된 그룹의 예를 알아보자. 누구나 사용하고 있는 컴퓨터의 운영체제를 살펴보자. 운영체제에서 사용자의 자료를 파일의 형태로 보관하고, 이러한 파일들을 그룹화 한 것을 폴더라고 한다.

파일과 폴더를 선형으로 구성할 경우와 비선형으로 구성하는 경우 각각 어떤 장, 단점이 있는지 생각해보자.



Windows의 탐색기

위 그림은 일반적으로 운영체제에서 파일과 폴더를 구조화하는 방법이다. 위 구조는 자료를 트리형태로 구성하여 탐색하기 쉽도록 구성한다는 사실을 알 수 있다.

만약 파일과 폴더를 선형으로 구성한다면 어떤 단점이 있을지 생각해보자.

그리고 위 구조에서 C드라이브에서 내가 원하는 파일을 찾고자 한다면 어떤 탐색법이 필요할지 생각해 보는 것도 알고리즘 설계능력을 키우는데 많은 도움이 된다. 단순한 반복문으로 모든 폴더를 방문하는 알고리즘을 설계하기는 쉽지 않다.

탐색기반설계는 주어진 문제에서 주어진 데이터를 특성에 맞도록 구조화하고 이 자료를 적절한 방법으로 탐색해 나가면서 원하는 해를 찾는 알고리즘 설계법이며 탐색하는 범위

에 따라서 크게 전체탐색법이라 하고, 탐색할 영역을 적절한 방법으로 배제하여 탐색의 효율을 높인 방법을 부분탐색법이라 한다.

이 단원에서는 전체탐색법과 부분탐색법으로 주어진 문제의 해를 구하는 방법에 대해서 학습한다.

가. 선형구조의 탐색

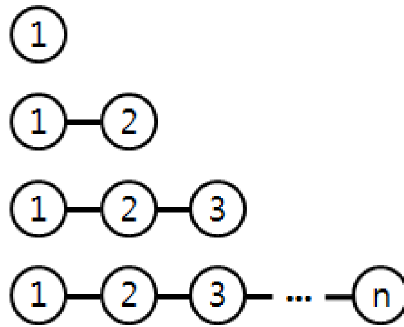
선형구조란 자료의 순서를 유일하게 결정할 수 있는 형태의 구조를 말한다. i번째 자료를 탐색한 다음, $i+1$번째로 탐색해야할 자료가 유일한 형태를 의미한다. 2차원, 3차원 구조라도 순서가 일정하게 정해져 있으면 이는 선형이라고 할 수 있다.

선형구조는 주로 배열과 리스트의 형태로 저장된다. 일반적으로 1차원 배열에 자료를 저장하는 1차원 선형구조와 2차원 이상의 배열에 자료가 저장이 되어있는 다차원 선형구조로 나눌 수 있다.

선형구조의 탐색은 선형구조로 저장된 자료들 중에서 원하는 것을 찾는 작업을 말한다. 선형구조를 탐색하는 방법은 기본적으로 순차탐색과 이분탐색이 있고, 이들을 적절히 응용한 탐색법도 만들어 사용할 수 있다. 이 단원에서는 순차탐색과 이분탐색을 익히고 이를 통하여 간단한 문제를 해결하는 실습을 한다.

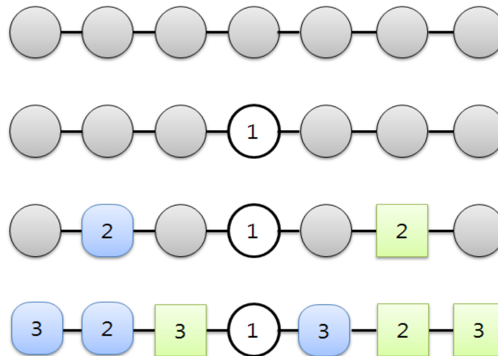
순차탐색은 자료의 특성에 관계없이 사용할 수 있는 일반적인 방법으로 전체탐색기법의 한 방법이다. 첫 번째 원소로부터 시작하여 한 원소씩 차례로 다음 원소를 탐색해 나가는 방법으로 자료가 n개 있을 때의 계산량은 $O(n)$이다.
`

탐색 순서를 그림으로 나타내면 다음과 같다.



순차탐색의 순서

다음 알고리즘은 배열 S에 n개의 원소를 입력받고, 그 중에 k가 있는지를 찾는 알고리즘이다. 이 알고리즘은 오름차순이나 내림차순으로 정렬된 선형구조에서 원하는 원소를 찾는 것으로 계산량은 $O(\log_2 n)$이다.



이분탐색의 탐색순서(○는 처음 접근하는 원소이고, 사각형은 찾은 곳의 값이 찾으려는 값보다 작으면 찾는 위치, 둥근 사각형은 그 값의 반대조건일 경우에 탐색하는 위치이다. 조건의 결과에 따라 왼쪽 또는 오른쪽 중 하나를 탐색하게 된다.)

다음은 이분탐색을 구현한 소스코드이다. 이를 재귀함수로도 구현할 수 있으므로 한 번 생각해보기 바란다.

```c
#include <stdio.h>

int S[100], n, k;

int find(int s, int e)
{
  while(s<=e)
  {
    int m=(s+e)/2;
    if(S[m]==k) return m;
    if(S[m]>k) e=m-1;
    else s=m+1;
  }
  return -1;
}

int main()
{
  scanf("%d%d", &n, &k);
  for(int i=0; i<n; i++ )
    scanf("%d", &S[i]);
  printf("%d\n", find(0, n-1));
  return 0;
}
```

기본적인 탐색방법을 익힐 수 있는 다음 문제들을 해결해보자.

문제 1

최댓값(S)

9개의 서로 다른 자연수가 주어질 때, 이들 중 최댓값을 찾고 그 값이 몇 번째 수인지를 구하는 프로그램을 작성하시오.

예를 들어, 서로 다른 9개의 자연수가 각각

3, 29, 38, 12, 57, 74, 40, 85, 61

라면, 이 중 최댓값은 85이고, 이 값은 8번째 수이다.

입력

첫째 줄부터 아홉째 줄까지 한 줄에 하나의 자연수가 주어진다. 주어지는 자연수는 100보다 작다.

출력

첫째 줄에 최댓값을 출력하고, 둘째 줄에 최댓값이 몇 번째 수인지를 출력한다.

입력 예	출력 예
3 29 38 12 57 74 40 85 61	85 8

출처: 한국정보올림피아드(2007 지역본선 초등부)

풀이

이 문제는 자료를 1차원 배열에 저장한 후 반복문을 이용하여 전체탐색법을 구현하면 쉽게 구할 수 있다. 전체탐색을 하더라도 탐색해야할 자료의 수가 9개뿐이므로 충분히 빠른 시간 내에 해를 구할 수 있는 기본적인 문제이다.

따라서 반복문을 구현하는 연습을 할 수 있는 문제로 이 문제를 해결하는 방법이 다른 문제들을 해결하는 도구로 많이 활용될 수 있으므로 꼭 익혀둘 수 있도록 한다.

일단 먼저 문제해결 아이디어를 생각하자. 최종적으로 출력할 해를 변수 ans로 두고, 최댓값의 인덱스를 저장할 변수를 index로 설정한다.

먼저 모든 자료를 탐색하기 전에 ans를 모든 원소들 보다 작은 값으로 설정한다. 이 문제에서는 100 이하의 자연수가 데이터의 정의역이므로, 0으로 설정하면 된다. 다음으로 첫 번째 자료부터 마지막 자료까지 하나씩 검사해가며 현재까지 ans보다 더 큰 값이 나타나면 ans를 갱신하고, index값도 갱신한다.

마지막 자료까지 탐색을 마치면, ans와 index를 출력하면 된다. 이 과정을 입출력 예를 통해서 알아보자.

준비 단계

A	3	29	38	12	57	74	40	85	61
index	0	1	2	3	4	5	6	7	8

현재 탐색 중인 index = ?　　ans　0

〈 입력받은 상태에서 탐색 준비를 한다. 탐색의 순서는 index가 0부터 8까지 〉

[1 단계]

A	3	29	38	12	57	74	40	85	61
index	0	1	2	3	4	5	6	7	8

현재 탐색 중인 index = 0 ans 3

〈A[0]의 값이 ans보다 크므로 ans의 값은 3이 된다.〉

[2 단계]

A	3	29	38	12	57	74	40	85	61
index	0	1	2	3	4	5	6	7	8

현재 탐색 중인 index = 1 ans 29

〈A[1]의 값이 현재 ans의 값 3보다 크므로 ans값은 29로 갱신〉

[3 단계]

A	3	29	38	12	57	74	40	85	61
index	0	1	2	3	4	5	6	7	8

현재 탐색 중인 index = 2 ans 38

〈A[2]의 값이 현재 ans의 값 29보다 크므로 ans값은 38로 갱신〉

[4 단계]

A	3	29	38	12	57	74	40	85	61
index	0	1	2	3	4	5	6	7	8

현재 탐색 중인 index = 3 ans 38

〈A[3]의 값이 현재 ans보다 작으므로 ans는 38을 유지〉

⋮

[9 단계]

A	3	29	38	12	57	74	40	85	61
index	0	1	2	3	4	5	6	7	8

현재 탐색 중인 index = 8　　ans　85

〈A[8]의 값이 현재 ans보다 작으므로 ans는 85를 유지〉

이와 같이 배열을 선형으로 전체탐색을 하면서 최댓값을 구할 수 있다. 이 방법은 가장 기본적인 방법 중 하나로 다른 알고리즘에 많이 응용되는 방법이므로 잘 익혀둘 수 있도록 한다. 이를 프로그램으로 구현하면 다음과 같다.

```
#include <stdio.h>
#define MAXN 9

int main()
{
  int i, ans, index, A[MAXN+1];
  ans = 0;
  index = 0;
  for( i=1; i<MAXN+1; i++)
    scanf("%d", &A[i]);
  for( i=1; i<MAXN+1; i++)
    if( ans<A[i] )
    {
      ans = A[i];
      index = i;
    }
  printf("%d\n%d\n", ans, index);
  return 0;
}
```

위 프로그램은 문제 1을 푼 가장 표준적인 코드라고 볼 수 있다.

자세한 내용을 설명하기에는 너무 길 수 있으므로 간단하게 말하자면, main 함수를 비롯한 모든 함수들은 함수 내에서 사용되는 모든 변수를 지역변수로 쓰는 것이 좋다. 즉 함수를 완전히 독립 모듈로 보더라도 모든 데이터는 모듈 내에서 정의되어야 한다는 의미이다. 만약 어떤 함수가 전역변수를 참조한다면, 이 함수를 다른 프로그램에 가져가면 전역변수를 더 이상 참조할 수 없으므로, 이는 좋은 모듈이라 할 수 없다.

하지만 정보올림피아드와 같은 대회에서는 주어진 짧은 시간에 알고리즘을 구현하는 것이 중요하다.

일반적으로 대회에 경험이 있는 학생들이 문제 1을 해결하는 데 구현하는 코드는 다음과 같다.

코드	참고

```
#include <stdio.h>
#define MAXN 9
int ans, index, A[MAXN+1];

void solve(void)
{
  for(int i=1; i<10; i++)
  {
    scanf("%d", &A[i]);
    if(ans<A[i]) ans=A[i], index=i;
  }
}

int main()
{
  solve();
  printf("%d\n%d\n", ans, index);
  return 0;
}
```

이 풀이에서는 일반적으로 좋지 않은 방법으로 보이는 코드들이 많이 보인다. ans, index를 모두 전역으로 선언하고 있으며, 초기화도 하지 않고 있다.

실제 대회에서는 이와 같이 전역변수를 활용하는 경우가 많다. 특히 배열을 선언하는 경우는 대부분 전역변수를 활용한다. 가장 큰 이유는 지역변수보다 더 많은 공간을 확보

할 수 있다는 장점이 있으며, 0으로 초기화되는 점도 무시할 수 없기 때문이다.

그리고 solve라는 모듈로 풀이를 분리하는 것도 자주 볼 수 있는데, 각종 대회에서는 디버깅을 빨리할 수 있는 능력이 매우 중요하다. 이와 같이 모듈을 입력, 처리, 출력으로 나눠두면 오류가 발생했을 때, 이를 처리하기 용이하다.

마지막으로 for문 내부에 i와 같은 반복문의 인덱스 변수를 선언한다는 점이다. 프로그램 전체적으로 i와 같은 변수를 자주 활용하게 되는데 이로 인한 오류가 의외로 많이 발생한다. 따라서 각 반복문별로 영역을 제한하여 사용하는 것도 일종의 오류를 줄이기 위한 팁이라고 할 수 있다.

중요한 것은 자기만의 코딩스타일을 구축하여 실수를 최소화하는 것이 매우 중요한 점이기 때문에 쉬운 문제들을 다룰 때부터 자신만의 습관을 기르는 방법을 익힐 수 있도록 하자.

마지막으로 다음과 같이 더 효율적으로 작성할 수도 있으니 참고하기 바란다.

```
#include <stdio.h>
#define MAXN 9
int ans, A[MAXN+1];

void solve(void)
{
  for(int i=1; i<10; i++)
  {
    scanf("%d", A+i);
    if(A[ans]<A[i]) ans=i;
  }
}

int main()
{
  solve();
  printf("%d\n%d\n", A[ans], ans);
  return 0;
}
```

이번 풀이에서는 ans, index를 하나의 변수 ans로 처리하고 있다. 그리고 9행에서 입력 받을 때 “&A[i]” 대신 “A+i”를 활용하고 있다. 이러한 코딩 스타일도 자주 활용되는 방법으로 배열과 포인터를 이해하면 위와 같이 사용할 수 있음을 알 수 있다. 이와 같을 때에는 특수문자로 인한 오타의 확률도 줄일 수 있으므로 다양한 방법을 익힐 수 있도록 하자.

그리고 이 문제에서는 배열을 쓰지 않고도 해결 가능하므로 다양한 방법으로 코딩 연습을 해두는 연습을 하자.

문제 2

3의 배수 게임

3의 배수 게임을 하던 정올이는 3의 배수 게임에서 잦은 실수로 계속해서 벌칙을 받게 되었다.

3의 배수 게임의 왕이 되기 위한 마스터 프로그램을 작성해 보자.

** 3의 배수 게임이란?

여러 사람이 순서를 정해 순서대로 수를 부르는 게임이다.
만약 3의 배수를 불러야 하는 상황이라면, 그 수 대신 "박수" 를 친다.

입력

첫 째 줄에 하나의 정수 n이 입력된다(n은 10미만의 자연수이다.).

출력

1부터 그 수까지 순서대로 공백을 두고 수를 출력하는데, 3 또는 6 또는 9인 경우 그 수 대신 영문 대문자 X를 출력한다.

입력 예	출력 예
7	1 2 X 4 5 X 7

풀이

이 문제도 [문제 1]과 마찬가지로 단순히 반복문을 이용하여 전체탐색법으로 해결할 수 있다. 단지 이 문제는 특정 값을 찾거나 하는 것이 아니라 전체 데이터를 읽으면서 특정 자료가 있으면 변경한다는 점은 다르나 전반적으로 같은 방법으로 해결할 수 있다.

이 문제에서 특정 자료란 입력된 숫자가 3의 배수일 경우를 말한다. 임의의 변수 n이 3의 배수인지 판정하는 가장 일반적인 방법은 다음과 같은 방법을 이용한다.

```
n % 3 == 0
```

또는 정수의 나눗셈의 특성을 이용한 다음과 같은 방법도 있다.

```
n / 3 * 3 == n
```

이 방법의 경우는 3의 배수가 아니면 3으로 나누어서 곱할 때 원래의 수가 되지 않는다. 다음 표는 1부터 10까지의 자연수를 3으로 나눈 후 곱할 때의 값을 나타낸다.

n	n/3	n/3*3==n
1	0	False
2	0	False
3	**1**	**True**
4	1	False
5	1	False
6	**2**	**True**
7	2	False
8	2	False
9	**3**	**True**
10	3	False

위의 방법들 이외에도 다양한 방법들이 있으므로 다양하게 생각해보기 바란다. 문제를 해결할 때, 1부터 n까지 1씩 증가하여 탐색하면서 각 수가 3의 배수인지 판정하여 3의 배

수이면 “X”를 아니면 그 수를 출력하도록 작성하면 쉽게 해결할 수 있다.

이 문제를 해결한 예시는 다음과 같다.

```
#include <stdio.h>

int n;

void solve(void)
{
  for(int i=1; i<=n; i++)
  {
    if(i%3==0) printf("X ");
    else printf("%d ", i);
  }
}

int main()
{
  scanf("%d", &n);
  solve();
  return 0;
}
```

이 문제에서도 입력값 n을 전역변수로 두고 있다. 이 방법에 대해서는 각자 자신의 방법을 정하기 바라며, solve 함수의 for문에서 i를 1부터 시작해야 한다는 점도 유의할 필요가 있다. 일반적으로 반복문의 0부터 출발하거나 1부터 출발하는데, 문제의 특성과 개인의 성향에 따라서 차이가 있다. 이 부분 또한 자신의 스타일을 정확하게 정해두면 실수를 줄일 수 있다.

앞으로 입출력에 특별한 사항이 없을 때에는 풀이는 다음과 같이 solve 함수 부분만 제시하는 경우도 있을 것이다. 이때는 전역변수와 main 함수는 solve 함수로 충분히 유추할 수 있을 것이다.

줄	코드	참고
1	`void solve(void)`	5: 두 번째 방법을 이용
2	`{`	
3	`  for(int i=1; i<=n; i++)`	
4	`  {`	
5	`    if(i/3*3==i) printf("X ");`	
6	`    else printf("%d ", i);`	
7	`  }`	
8	`}`	

문제 3

linear structure search

n개로 이루어진 정수 집합에서 원하는 수의 위치를 찾으시오.

단, 입력되는 집합은 오름차순으로 정렬되어 있으며, 같은 수는 없다.

입력

첫 줄에 한 정수 n이 입력된다.
둘째 줄에 n개의 정수가 공백으로 구분되어 입력된다.
셋째 줄에는 찾고자 하는 수가 입력된다.
(단, 2 <= n <= 1,000,000 , 각 원소의 크기는 100,000,000을 넘지 않는다.)

출력

찾고자 하는 원소의 위치를 출력한다. 없으면 -1을 출력한다.

입력 예	출력 예
8 1 2 3 5 7 9 11 15 11	7
3 2 5 7 3	-1

풀이

이 문제는 앞에서 다룬 이분탐색의 예제 프로그램을 거의 그대로 활용할 수 있는 문제이다. 일단 탐색 범위를 [s, e]로 정한 다음 범위의 가운데 위치의 값을 m으로 설정하고 탐색을 시작한다.

배열을 A라고 할 때, A[m] == k 인 경우와 A[m] 〉 k, A[m] 〈 k인 경우로 나누어 처리하는 방법으로 문제를 해결할 수 있다. 자세한 과정은 다음과 같다.

준비 단계

A	1	2	3	5	7	9	11	15
index	0	1	2	3	4	5	6	7

k		s		e		m	
	11		0		7		?

〈 입력받은 상태에서 탐색 준비를 한다. 탐색 범위는 0~7 〉

[1 단계]

A	1	2	3	5	7	9	11	15
index	0	1	2	3	4	5	6	7

k		s		e		m	
	11		0		7		3

〈 A[3]의 값이 k보다 작으므로, 영역을 4~7로 변경 〉

[2 단계]

A	1	2	3	5	7	9	11	15
index	0	1	2	3	4	5	6	7

k		s		e		m	
	11		4		7		5

〈 A[5]의 값이 k보다 작으므로, 영역을 6~7로 변경 〉

[3 단계]

A	1	2	3	5	7	9	11	15
index	0	1	2	3	4	5	6	7

k	11	s	6	e	7	m	6

〈A[6]의 값과 k가 같으므로 탐색종료, 찾은 인덱스는 6〉

위 방법을 소스코드로 작성하면 다음과 같다.

줄	코드	참고
1	`#include <stdio.h>`	
2	`int n, k, A[1000001];`	
3	`int solve(int s, int e)`	
4	`{`	
5	`  int m;`	
6	`  while(e-s>=0)`	
7	`  {`	
8	`    m=(s+e)/2;`	
9	`    if(A[m]==k)`	
10	`      return m+1;`	
11	`    if(A[m]<k) s=m+1;`	
12	`    else e=m-1;`	
13	`  }`	
14	`  return -1;`	
15	`}`	
16	`int main()`	
17	`{`	
18	`  scanf("%d",&n);`	
19	`  for(int i=0; i<n; i++ )`	
20	`    scanf("%d", A+i);`	
21	`  scanf("%d",&k);`	
22	`  printf("%d\n", solve(0, n-1));`	
23	`  return 0;`	
24	`}`	

앞의 예제에서는 5행을 e >= s로 작성했으나 일반화를 위하여 e-s >= 0 으로 활용했으며, 나머지 이분탐색 부분들도 자세히 분석하여 익힐 수 있도록 한다.

위 소스코드를 다음과 같은 재귀함수로도 만들 수 있다. 재귀함수는 매우 다양한 응용이 가능하므로 이해해두면 많은 도움이 된다.

```
#include <stdio.h>
int n, k, A[1000001];
int solve(int s, int e)
{
  if(s>e)
    return -1;
  int m=(s+e)/2;
  if(A[m]==k)
    return m+1;
  if(A[m]<k)
    return solve(m+1, e);
  else
    return solve(s, m-1);
}
int main()
{
  scanf("%d",&n);
  for(int i=0; i<n; i++ )
    scanf("%d", A+i);
  scanf("%d",&k);
  printf("%d\n", solve(0, n-1));
  return 0;
}
```

문제 4

lower bound

n개로 이루어진 정수 집합에서 원하는 수 k 이상인 수가 처음으로 등장하는 위치를 찾으시오.

단, 입력되는 집합은 오름차순으로 정렬되어 있으며, 같은 수가 여러 개 존재할 수 있다.

입력

첫째 줄에 한 정수 n이 입력된다.
둘째 줄에 n개의 정수가 공백으로 구분되어 입력된다.
셋째 줄에는 찾고자 하는 값 k가 입력된다.

(단, 2 <= n <= 1,000,000 , 각 원소의 크기는 100,000,000을 넘지 않는다.)

출력

찾고자 하는 원소의 위치를 출력한다. 만약 모든 원소가 k보다 작으면 n+1을 출력한다.

입력 예	출력 예
5 1 3 5 7 7 7	4
8 1 2 3 5 7 9 11 15 6	5
5 1 2 3 4 5 7	6
5 2 2 2 2 2 1	1

풀이

이 문제에서 다루는 lower bound는 대회에 자주 등장하는 방법이므로 꼭 익혀둘 수 있도록 한다. lower bound는 이분탐색을 이용하여 구할 수 있다. 이분탐색은 찾고자 하는 값이 없으면 탐색 실패가 되지만, lower bound는 찾고자하는 정확한 값이 없더라도 찾고자 하는 값보다 큰 가장 작은 정수 값을 찾으므로 차이가 있다.

lower bound인 경우에는 같은 원소가 여러 개 있더라도 항상 유일한 해를 구할 수 있기 때문에 알고리즘을 설계하는 것이 이분탐색 보다는 까다로우나 근본은 같으므로 잘 익혀 둘 수 있도록 한다.

먼저 구간을 [s, e]로 설정하고, 중간위치의 값을 m이라 하면, A[m-1] 〈k 이면서 A[m]〉= k인 최소 m을 찾는 문제가 된다. 이 때 m은 2이상인 값이다. 따라서 일반적인 이분탐색에서 A[m] == k인 부분을 다른 부분에 포함해야 한다는 점을 잘 확인해야 한다.

다음으로 모든 원소가 k보다 작을 때는 n+1을 출력해야 하므로 처음 구간을 잡을 때, [1, n]을 잡는 것이 아니라 [1, n+1]로 설정하여 시작한다는 점도 유의해야 한다.

준비 단계

A	1	2	3	5	7	9	11	15
index	0	1	2	3	4	5	6	7

k	s	e	m
6	0	8	?

〈 입력받은 상태에서 탐색 준비를 한다. 탐색 범위는 0~7 〉

[1 단계]

A	1	2	3	5	7	9	11	15
index	0	1	2	3	4	5	6	7

k	6	s	0	e	8	m	4

〈A[4]가 6보다 크므로 범위를 0~4로 한다. 만약 일반 이분탐색이었으면 0~3으로 범위를 좁혀야 하나 lower bound는 k 이상이 최솟값의 위치이므로 e까지 포함한다.〉

[2 단계]

A	1	2	3	5	7	9	11	15
index	0	1	2	3	4	5	6	7

k	6	s	0	e	4	m	2

〈A[2]가 6보다 작으므로 범위를 3~4로 하고 재탐색을 시작한다.〉

[3 단계]

A	1	2	3	5	7	9	11	15
index	0	1	2	3	4	5	6	7

k	6	s	3	e	4	m	3

〈A[3]이 6보다 작으므로 범위를 4 ~ 4로 하고 재탐색을 시작한다.〉

[4 단계]

A	1	2	3	5	7	9	11	15
index	0	1	2	3	4	5	6	7

k	6	s	4	e	4	m	4

〈이제 더 이상 탐색할 원소가 없으므로 인덱스 4에 있는 원소가 k 이상인 최소 원소의 위치가 된다.〉

이와 같이 lower bound는 이분탐색과 유사하나 좀 더 엄밀하게 접근해야 한다. 그리고 매우 다양한 응용범위가 있으므로 잘 익힐 수 있도록 한다.

위의 과정을 구현한 소스코드는 다음과 같다.

줄	코드	참고

```
1  #include <stdio.h>
2  int n, k, A[1000001];
3  int solve(int s, int e)
4  {
5    int m;
6    while(e-s>0)
7    {
8      m=(s+e)/2;
9      if(A[m]<k) s=m+1;
10     else e=m;
11   }
12   return e+1;
13 }
14 int main()
15 {
16   scanf("%d",&n);
17   for(int i=0; i<n; i++)
18     scanf("%d", A+i);
19   scanf("%d",&k);
20   printf("%d\n", solve(0, n));
21   return 0;
22 }
```

6행의 e-s 〉 0 은 e 〉 s와 같은 의미이다. while문을 탈출했을 때, 즉 11행에서는 s 〉= e가 된다는 사실을 잘 이해하고 있어야 한다. 따라서 12행에서 반환하는 값이 e+1이 된다는 것이 중요한 점이다.

만약 6행을 e-s 〉 1로 설정한다면 어떻게 될지도 생각해보면 실력향상에 많은 도움이 될 것이다. 이러한 부분들은 수학적으로 엄밀하게 접근하는 연습을 하는 데 많은 도움이 되니 꼭 실습해보기 바란다.

이번에는 이 문제를 해결하는 데 STL을 직접 활용하는 방법을 소개한다. 사실 실제 대회에서는 이렇게 lower bound를 작성하는 경우는 흔치 않으며, 대부분 std::lower_bound( )

함수를 활용하게 될 것이므로 이 std::lower_bound( )를 꼭 익힐 수 있도록 한다.

S라는 배열의 처음부터 n-1번째까지의 원소들 중 k의 low bound에 해당하는 원소의 위치를 반환하는 std::lower_bound( )의 기본적인 사용법은 아래와 같다.

```
std::lower_bound( S, S+n, k, [compare] );
```

여기서 compare 함수는 앞에 std::sort()에서 사용했던 compare와 같은 역할을 하는 함수로서 작성법도 동일하므로, 앞에 예를 참고하면 된다. 그리고 compare를 생략할 경우에는 오름차순이라고 가정하고 동작하게 된다.

다음 소스코드는 std::lower_bound()를 활용하여 문제를 해결한 것을 보여준다. 이 예는 자주 활용할 가능성이 크므로 반드시 익혀두기 바란다.

줄	코드	참고

```
1   #include <stdio.h>
2   #include <algorithm>
3
4   int n, k, A[1000001];
5
6   int main()
7   {
8     scanf("%d",&n);
9     for(int i=0; i<n; i++)
10      scanf("%d", A+i);
11    scanf("%d",&k);
12    printf("%d\n", std::lower_bound(A,A+n,k)-A+1);
13  }
```

이 소스코드에서는 12행의 내용을 이해하는 것이 중요하다.

std::lower_bound(A, A+n, k)의 의미는 배열 A[0]~A[n-1]이 오름차순으로 정렬되어 있을 때, k의 lower bound위치의 주소를 구한다.

따라서 그 주소에서 A를 빼면 k가 존재하는 배열 A의 인덱스가 되며, 배열의 인덱스는 0부터 시작하므로 1을 더해주면 우리가 원하는 해를 구할 수 있게 된다.

따라서 std::lower_bound(A, A+n, k)-A+1과 같이 활용할 수 있다.

문제 5

upper bound

n개로 이루어진 정수 집합에서 원하는 수 k보다 큰 수가 처음으로 등장하는 위치를 찾으시오.

단, 입력되는 집합은 오름차순으로 정렬되어 있으며, 같은 수가 여러 개 존재할 수 있다.

입력

첫째 줄에 한 정수 n, 둘째 줄에 n개의 정수가 공백으로 구분되어 입력된다. 셋째 줄에는 찾고자 하는 값 k가 입력된다.
(단, 2 <= n <= 1,000,000, 각 원소의 크기는 100,000,000을 넘지 않는다.)

출력

찾고자 하는 원소의 위치를 출력한다. 만약 모든 원소가 k보다 작으면 n+1을 출력한다.

입력 예	출력 예
5 1 3 5 5 7 5	5
8 1 2 7 7 7 7 11 15 7	6
5 1 2 3 4 5 7	6
5 2 2 2 2 2 1	1

풀이

이 문제에서 다루는 upper bound 또한 대회에 자주 등장하는 방법이므로 꼭 익혀둘 수 있도록 한다. upper bound도 lower bound와 마찬가지로 이분탐색을 이용하여 구할 수 있다. upper bound는 k를 초과하는 가장 첫 번째 원소의 위치를 구하는 것이다.

upper bound 도 lower bound와 같이 같은 원소가 여러 개 있더라도 항상 유일한 해를 구할 수 있기 때문에 알고리즘을 설계하는 것이 이분탐색 보다는 까다롭다.

하지만 upper bound와 lower bound를 함께 이용하면 다양한 문제를 접근할 수 있다. 예를 들어 정렬된 배열에 존재하는 k는 모두 몇 개인가? 와 같은 문제도 위 두 함수를 이용하면 쉽게 해결할 수 있으므로 이러한 문제에 대해서도 따로 연습해 둘 필요가 있다.

upper bound를 구하기 위해서는 먼저 구간을 [s, e]로 설정하고, 중간위치의 값을 m이라 하면, A[m-1] 〈= k 이면서 A[m] 〉 k인 최소 m을 찾는 문제가 된다. 이 때 m은 2이상인 값이다. 따라서 일반적인 이분탐색에서 A[m] == k인 부분을 다른 부분에 포함해야 한다는 점을 잘 확인해야 한다.

다음으로 모든 원소가 k보다 작을 때는 n+1을 출력해야 하므로 처음 구간을 잡을 때, [1, n]을 잡는 것이 아니라 [1, n+1]로 설정하여 시작한다는 점도 유의해야 한다. 다음은 upper bound의 과정을 나타낸다.

준비 단계

A	1	2	7	7	7	7	11	15
index	0	1	2	3	4	5	6	7

k	s	e	m
7	0	8	?

〈 입력받은 상태에서 탐색 준비를 한다. 탐색 범위는 0~7 〉

[1 단계]

A	1	2	7	7	7	7	11	15
index	0	1	2	3	4	5	6	7

k 7 s 0 e 8 m 4

〈A[4]와 7이 같으므로 범위를 5~8로 설정한다. 만약 이분탐색이었으면 바로 탐색을 종료해야 하나 upper bound는 k를 초과하는 최솟값의 위치이므로 4를 포함할 필요가 없다.〉

[2 단계]

A	1	2	7	7	7	7	11	15
index	0	1	2	3	4	5	6	7

k 7 s 5 e 8 m 6

〈A[6]이 7보다 크므로 범위를 5~6까지로 하고 재탐색을 시작한다.〉

[3 단계]

A	1	2	7	7	7	7	11	15
index	0	1	2	3	4	5	6	7

k 7 s 5 e 6 m 5

〈A[5]와 7이 같으므로 범위를 6~6까지로 하고 재탐색을 시작한다.〉

[4 단계]

A	1	2	7	7	7	7	11	15
index	0	1	2	3	4	5	6	7

k	s	e	m
7	6	6	6

〈범위 상 더 이상 탐색을 할 필요가 없으므로 6번 인덱스가 조건을 만족하는 가장 작은 인덱스라는 사실을 확인할 수 있음.〉

이와 같이 upper bound는 lower bound와 유사하다. 자세한 내용은 다음 소스코드를 참고한다.

줄	코드	참고

```
1  #include <stdio.h>
2  int n, k, A[1000001];
3  int solve(int s, int e)
4  {
5    int m;
6    while(e-s>0)
7    {
8      m = (s+e)/2;
9      if(A[m]<=k) s=m+1;
10     else e=m;
11   }
12   return e+1;
13 }
14 int main()
15 {
16   scanf("%d",&n);
17   for(int i=0; i<n; i++)
18     scanf("%d", A+i);
19   scanf("%d",&k);
20   printf("%d\n", solve(0, n));
21 }
```

lower bound를 구할 때와 차이점은 9행에서 A[m] 〈 k를 A[m] 〈= k로 바꾼 것뿐이다. 이 부분을 그냥 단순히 외우려 하지 말고, 왜 부등식이 위와 같이 바뀌면 upper

bound를 구할 수 있는지 수학적으로 엄밀히 분석해 두면 나중에 다른 문제들을 해결할 때 많은 도움이 될 것이다.

이 문제 또한 6행을 e-s 〉 1로 설정한다면 어떻게 될지도 생각해보면 실력향상에 많은 도움이 될 것이다.

upper bound도 lower bound와 마찬가지로 STL을 활용할 수 있는 방법이 있다.

S라는 배열의 처음부터 n-1번째까지의 원소들 중 k의 upper bound에 해당하는 원소의 주소를 반환하는 std::upper_bound()의 기본적인 사용법은 아래와 같다.

```
std::upper_bound( S, S+n, k, [compare] );
```

여기서 compare함수는 앞에 std::sort()에서 사용했던 compare와 같은 역할을 하는 함수고 작성법도 동일하므로, 앞의 예를 참고하면 된다. 그리고 compare를 생략할 경우에는 오름차순이라고 가정하고 동작하게 된다.

다음 소스코드는 std::upper_bound()를 활용하여 문제를 해결한 것을 보여준다. 이 예는 자주 활용할 가능성이 크므로 반드시 익혀두기 바란다.

줄 | 코드 | 참고

```
#include <stdio.h>
#include <algorithm>

int n, k, A[1000001];

int main()
{
  scanf("%d",&n);
  for(int i=0; i<n; i++)
    scanf("%d", A+i);
  scanf("%d",&k);
  printf("%d\n", std::upper_bound(A, A+n, k)-A+1);
}
```

이 소스코드에서는 12행의 내용을 이해하는 것이 중요하다.

std::upper_bound(A, A+n, k)의 의미는 배열 A[0]~A[n-1]이 오름차순으로 정렬되어 있을 때, k의 upper bound위치의 주소를 구한다.

주소에 대한 설명은 lower bound에서 설명한 것과 같다. 따라서 std::upper_bound(A, A+n, k)-A+1 와 같이 활용할 수 있다.

나. 비선형구조의 탐색

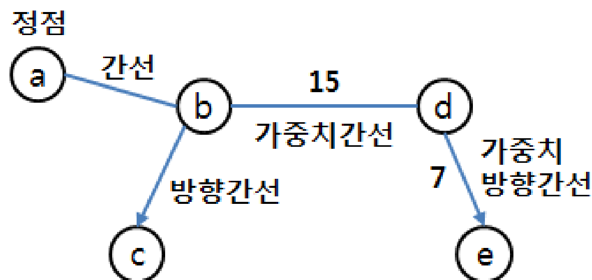
비선형구조란 *i* 번째 원소를 탐색한 다음 그 원소와 연결된 다른 원소를 탐색하려고 할 때, 여러 개의 원소가 존재하는 탐색구조를 말한다. 일반적으로 자료가 트리나 그래프로 구성되어 있을 경우를 비선형구조라 하고 이러한 트리나 그래프의 모든 정점을 탐색하는 것을 비선형 탐색이라고 이해하면 된다.

비선형구조는 선형과 달리 자료가 순차적으로 구성되어 있지 않으므로 단순히 반복문을 이용하여 탐색하기에는 어려움이 있다. 그러므로 비선형구조는 스택이나 큐와 같은 자료구조를 활용하여 탐색하는 것이 일반적이다.

비선형구조의 탐색은 크게 깊이우선탐색(depth first search, dfs)과 너비우선탐색(breadth firth search, bfs)으로 나눌 수 있으며, 이 두 가지 탐색법을 활용한 다양한 응용이 있으나 이 교재에서는 기본적인 두 가지 탐색법에 대해서 익히도록 한다.

- 비선형구조

비선형구조의 탐색을 다루기 전에 그래프와 트리에 대해서 간단히 알아보자. 트리와 그래프를 이루는 기본 요소를 정점(vertex)과 간선(edge)이라고 한다.

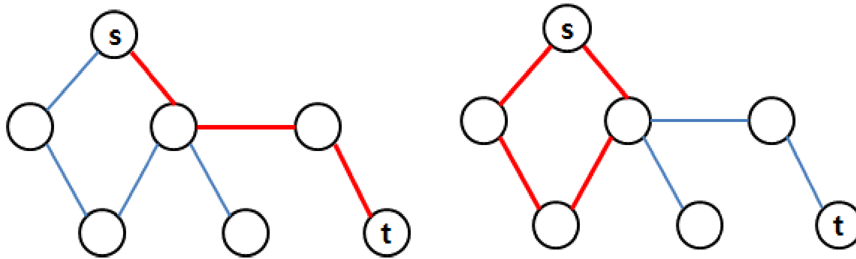


원은 정점, 선분은 간선을 나타내며, $\overline{ab}$는 보통간선, $\overrightarrow{bc}$는 방향간선, $\overline{bd}$는 가중치가 15인 양방향통행 간선, $\overrightarrow{de}$는 가중치가 7인 일방통행 간선(방향간선)을 나타낸다.

정점은 점 또는 원으로 표현하며, 일반적으로 상태나 위치를 표현한다. 간선은 정점들을 연결하는 선으로 표현하며, 정점들 간의 관계를 표현한다.

- 경로(path)와 회로(cycle)

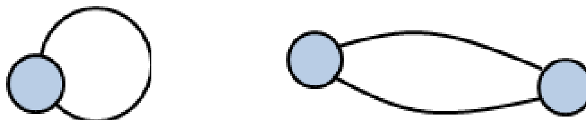
그래프에서 임의의 정점 s에서 임의의 정점 t로 이동할 때, s에서 t로 이동하는데 사용한 정점들을 연결하고 있는 간선들의 순서로 된 집합을 경로라고 한다. 회로는 그래프에서 임의의 정점 s에서 같은 정점 s로의 경로들을 말한다.



왼쪽의 빨간 간선들은 s에서 t로의 경로를 나타내고, 오른쪽의 빨간 간선들은 s에서 t로의 회로를 나타낸다.

- 자기간선(loop)과 다중간선(multi edge)

임의의 정점에서 자기 자신으로 연결하고 있는 간선을 자기간선, 임의의 정점에서 다른 정점으로 연결된 간선의 수가 2개 이상일 경우를 다중간선이라고 한다.

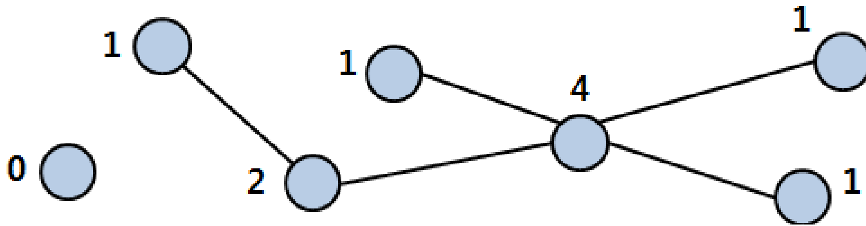


왼쪽은 자기간선 오른쪽은 다중간선을 나타낸다.

- 그래프의 차수(degree)

그래프의 임의의 한 정점에서 다른 정점으로 연결된 간선의 수를 차수라고 한다. 다음

그림은 각 정점에 대한 차수를 나타낸다.



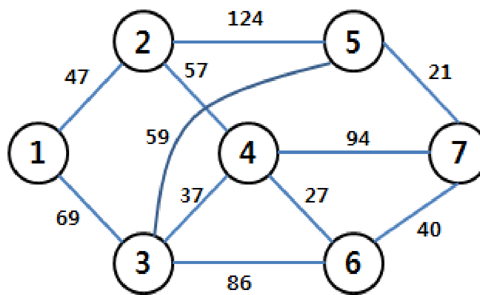
각 정점에서의 차수

– 그래프의 구현

그래프를 구현하는 방법은 인접행렬(adjacency matrix)과 인접리스트(adjacency list)로 크게 나눌 수 있다.

일반적으로 정보올림피아드를 비롯한 프로그래밍 대회에서 그래프는 정점의 수, 간선의 수, 각 간선들이 연결하고 있는 정점 2개로 이루어진 정보가 주어지는 경우가 대부분이다.

다음은 실제로 그래프가 주어질 때, 이를 저장하는 2가지 방법을 보여준다.



7개의 정점과 11개의 간선을 가지는 가중치 그래프의 예

이러한 그래프의 경우 일반적인 입력데이터의 형식은 다음과 같다.

[표-2] 그래프의 대표적인 입력형식과 입력데이터의 예

입력 형식	입력데이터의 예
첫 번째 줄에 정점의 수 n과 간선의 수 m이 공백으로 구분되어 입력된다. 두 번째 줄부터 m개의 줄에 걸쳐서 간선으로 연결된 두 정점의 번호와 가중치가 공백으로 구분되어 입력된다.	7 11 1 2 47 1 3 69 2 4 57 2 5 124 3 4 37 3 5 59 3 6 86 4 6 27 4 7 94 5 7 21 6 7 40

– 인접행렬의 구현

[표-2]의 입력예시를 인접행렬로 받기 위해서는 2차원 배열을 이용한다. 먼저 최대 정점의 수에 맞추어 2차원 배열을 선언하고 각 배열의 칸에 연결된 정보를 저장한다. 앞 그래프를 2차원 행렬을 이용하여 다음과 같이 저장한다.

	1	2	3	4	5	6	7
1		1	1				
2	1			1	1		
3	1			1	1	1	
4		1	1			1	1
5		1	1				1
6			1	1			1
7				1	1	1	

	1	2	3	4	5	6	7
1		47	69				
2	47			57	124		
3	69			37	59	86	
4		57	37			27	94
5		124	59				21
6			86	27			40
7				94	21	40	

왼쪽은 가중치가 없는 표현, 오른쪽은 가중치가 있는 표현이다. 예를 들어, 3행 4열의 경우 왼쪽은 1, 오른쪽은 37이 기록되어 있다. 왼쪽의 1은 간선이 있음을 의미하고, 오른쪽은 간선이 있을 때 가중치를 저장한다.

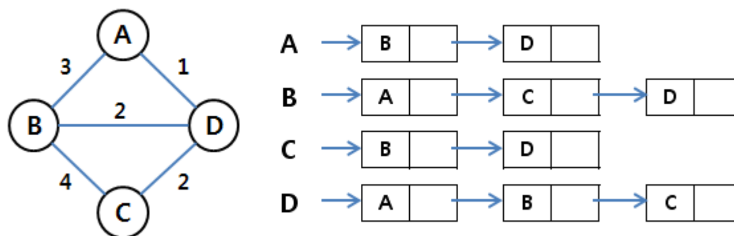
2차원 행렬을 이용하여 저장하는 소스코드는 다음과 같다. 단 최대 정점의 수는 100개로 가정한다.

줄	코드	참고
1	#include <stdio.h>	10: 정점 a, b를 연결하는 간선, w는 가중치 12: 만약 가중치가 없다면 1,방향간선이면G[a][b]만 저장.
2		
3	int n, m, G[101][101];	
4		
5	int main()	
6	{	
7	scanf("%d %d",&n,&m);	
8	for(int i=0; i<m; i++)	
9	{	
10	int a, b, w;	
11	scanf("%d %d %d", &a, &b, &w);	
12	G[a][b]=G[b][a]=w;	
13	}	
14	}	

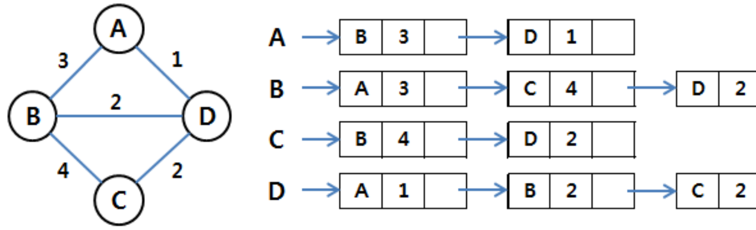
– 인접리스트의 구현

인접행렬로 표현할 때에는 연결되지 않았던 부분까지 모두 표현이 된다. 즉, 각 칸에 0이라고 기록된 부분은 연결이 되지 않은 부분을 의미한다. 사실 일반적인 그래프에서 행렬 상에서 0이라고 표현되는 부분이 많을 가능성이 크다.

알고리즘을 구현할 때에도 이 0이라고 표시된 부분까지 모두 조사를 해야 하므로 효율이 떨어지는 경우가 많다. 이러한 단점을 극복하기 위하여 제안된 방법이 인접리스트이고 이 방법은 인접행렬에서 0으로 표시된 부분은 저장하지 않으므로 효율을 높이고 있다.



[그림-10] 그래프의 인접리스트 표현



[그림-11] 가중치 그래프의 인접리스트 표현

[표-2]의 입력 예시를 인접리스트로 구현하기 위해서는 [그림-10], [그림-11]과 같이 연결리스트로 구현할 수 있지만 STL에서 제공하는 std::vector()를 이용하여 간단하게 구현할 수 있다. [표-2]의 입력예시를 인접리스트로 구현하면 다음과 같은 그림이 된다.

1	2	47	3	69				
2	1	47	4	57	5	124		
3	1	69	4	37	5	59	6	86
4	2	57	3	37	6	27	7	94
5	2	124	3	59	7	21		
6	3	86	4	27	7	40		
7	4	94	5	21	6	40		

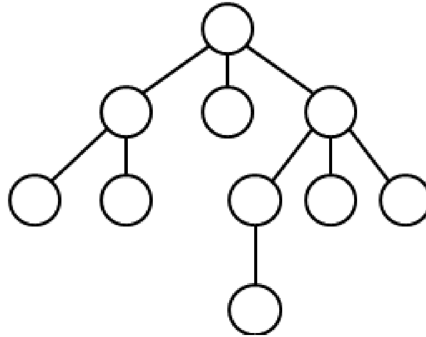
인접리스트에서는 정점과 가중치의 쌍으로 간선이 있는 것만 연결한다.
예를 들어 1행의 경우 1-2로의 간선의 가중치는 47이고 1-3으로의 간선의 가중치는 69라는 의미이다.

std::vector()를 이용한다면 위와 같이 인접행렬로 구현하는 것보다 공간을 적게 사용한다. 따라서 전체탐색법을 구현할 때, 당연히 탐색시간도 줄일 수 있다. 계산량으로 표현하자면, 인접행렬로 모든 정점을 탐색하는데 $O(nm)$의 시간이 드는데 반해, 인접리스트로 표현하면 $O(n+m)$의 시간이 든다.

여러 가지 장점이 있기 때문에 대회에서는 주로 인접리스트를 이용한 방법을 활용하는 경우가 많으므로 반드시 익혀둘 수 있도록 한다.

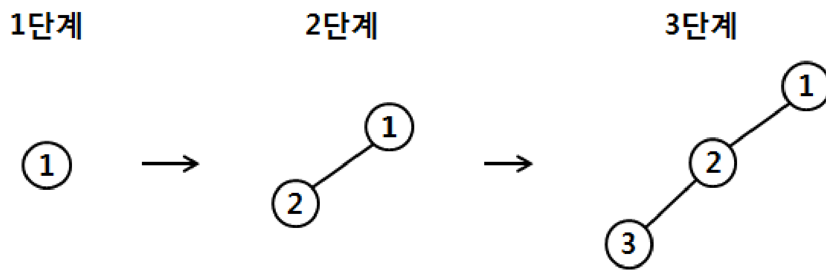
- 깊이우선탐색(dfs)

그래프 중 회로(cycle)가 없는 그래프를 트리라고 한다. [그림-13]은 트리를 나타낸다. 이 트리의 가장 위에 있는 정점에서 출발하여 모든 정점들을 깊이우선으로 탐색하며, 탐색하는 순서를 알아보자.



10개의 정점(vertex)과 9개의 간선(edge)을 가진 트리

출발 정점을 트리의 가장 위에 있는 정점으로 하고, 한 정점에서 이동 가능한 정점이 여러 개 있을 경우 왼쪽의 정점부터 방문한다고 가정하면, 단계별 탐색 과정은 다음과 같다.

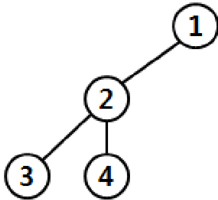


깊이우선 1~3 단계

깊이우선탐색과정에서 3단계 이후 더 이상 진행할 수 있는 정점이 없다. 그 이유는 간선으로 연결된 정점들 중 아직 방문하지 않은 정점을 방문하기 때문이다.

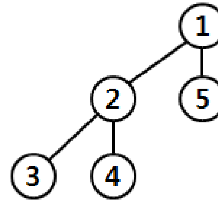
이처럼 더 이상 진행할 수 없을 때는 다시 이전 정점으로 되돌아가는 과정이 필요하다. 일반적으로 이 과정을 백트랙(backtrack)이라고 한다. 백트랙은 비선형구조의 탐색에서 매우 중요하다. 백트랙은 스택(stack)이나 재귀함수(recursion)를 이용하면 쉽게 구현할 수 있다.

4단계(backtrack)



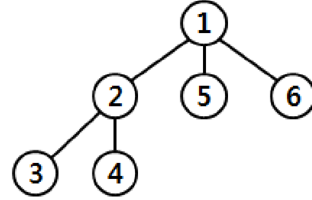
→

5단계(backtrack)



→

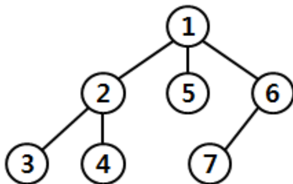
6단계(backtracking)



깊이우선 4 ~ 6 단계

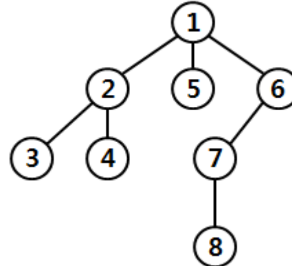
4, 5, 6단계는 연속으로 백트랙이 발생한다. 이는 더 이상 진행할 수 없는 정점까지 도달했다는 것을 의미한다. 계속 해서 다음 단계로 진행하는 과정은 다음과 같다.

7단계



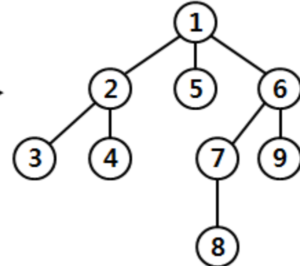
→

8단계



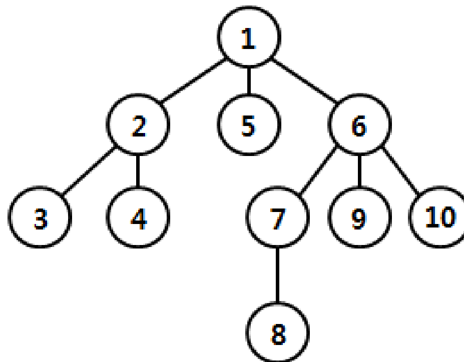
→

9단계(backtracking)



깊이우선 7~9 단계

위 단계에서 마지막 정점을 방문하면 깊이우선탐색이 완료된다.



탐색종료

깊이우선탐색을 정리하여 설명하면 먼저 시작 정점에서 간선을 하나 선택하여 진행할 수 있을 때까지 진행하고 더 이상 진행할 수 없다면 백트랙하여 다시 다른 정점으로 진행하여 더 이상 진행할 정점이 없을 때까지 이 과정을 반복하는 탐색법으로, 간선으로 연결된 모든 정점을 방문할 수 있는 탐색법이다.

깊이우선탐색의 알고리즘은 다음과 같다. 이 탐색법은 백트래킹(backtracking)이라는 알고리즘 설계 기법의 중심이 되며 백트래킹 기법은 모든 문제를 해결할 수 있는 가장 기본적인 방법이므로 꼭 익혀둘 필요가 있다.

줄	코드	참고
1 2 3 4 5 6 7 8 9 10 11	`bool visited[101];` `void dfs(int k)` `{` `  for(int i=0; i<G[i].size(); i++)` `    if(!visited[G[k][i].to])` `    {` `      visited[G[k][i].to]=true;` `      dfs(G[k][i]);` `    }` `  return;` `}`	1: 방문했는지 체크해 두는 배열 4: 정점 k와 연결된 모든 정점 방문 5: 만약 아직 방문하지 않았으면 7: 방문했다고 체크하고 8: 깊이우선탐색 진행 10: 더 이상 갈 길이 없으면 backtrack

이 방법은 그래프를 인접리스트에 저장했을 경우에 활용할 수 있다. 이 방법은 전체를 탐색하는데 있어서 반복문의 실행횟수는 모두 m 번이 된다. 따라서 일반적으로 속도가 더 빠르기 때문에 자주 활용된다.

만약 인접행렬로 그래프를 저장했다면 다음과 같이 작성하면 된다.

줄	코드	참고
1 2 3 4 5 6 7 8 9 10 11	`bool visited[101];` `void dfs(int k)` `{` `  for(int i=1; i<=n; i++)` `    if( G[k][i] && !visited[G[k][i]] )` `    {` `       visited[G[k][i]] = true;` `       dfs(G[k][i]);` `    }` `  return;` `}`	1: 방문했는지 체크해 두는 배열 4: 모든 정점에 대해서 검사 5: k에 연결되어 있으면서, 아직 방문하지 않았으면 7: 방문했다고 체크하고 8: 깊이우선탐색 진행 10: 더 이상 갈 길이 없으면 backtrack

이 방법은 전체를 탐색하는데 있어서 반복문을 n^2번 실행하게 된다. 따라서 평균적으로 인접리스트보다 느리지만 구현이 간편하므로, n값이 크지 않은 문제라면 충분히 적용할 가치가 있다.

문제 1

두더지 굴(S)

정올이는 땅속의 굴이 모두 연결되어 있으면 이 굴은 한 마리의 두더지가 사는 집이라는 사실을 발견하였다.

정올이는 뒷산에 사는 두더지가 모두 몇 마리인지 궁금해졌다. 정올이는 특수 장비를 이용하여 뒷산의 두더지 굴을 모두 나타낸 지도를 만들 수 있었다.

이 지도는 직사각형이고 가로 세로 영역을 0 또는 1로 표현한다. 0은 땅이고 1은 두더지 굴을 나타낸다. 1이 상하좌우로 연결되어 있으면 한 마리의 두더지가 사는 집으로 정의할 수 있다.

0	1	1	0	1	0	0
0	1	1	0	1	0	1
1	1	1	0	1	0	1
0	0	0	0	1	1	1
0	1	0	0	0	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0

[그림 1]

0	1	1	0	2	0	0
0	1	1	0	2	0	2
1	1	1	0	2	0	2
0	0	0	0	2	2	2
0	3	0	0	0	0	0
0	3	3	3	3	3	0
0	3	3	3	0	0	0

[그림 2]

[그림 2]는 [그림 1]을 두더지 굴로 번호를 붙인 것이다. 특수촬영 사진 데이터를 입력받아 두더지 굴의 수를 출력하고, 각 두더지 굴의 크기를 오름차순으로 정렬하여 출력하는 프로그램을 작성하시오.

입력

첫 번째 줄에 가로, 세로의 크기를 나타내는 n이 입력된다.(n은 30이하의 자연수)
두 번째 줄부터 n줄에 걸쳐서 n개의 0과 1이 공백으로 구분되어 입력된다.

출력

첫째 줄에 두더지 굴의 수를 출력한다. 둘째 줄부터 각 두더지 굴의 크기를 내림차순으로 한 줄에 하나씩 출력한다.

입력 예	출력 예
7 0 1 1 0 1 0 0 0 1 1 0 1 0 1 1 1 1 0 1 0 1 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0 1 1 1 1 1 0 0 1 1 1 0 0 0	3 9 8 7

풀이

이 문제는 그냥 보기에는 비선형탐색, 즉 그래프의 문제로 보이지 않는다. 하지만 지도에서 각 칸을 정점으로 생각하고 각 칸 중 1인 칸을 중심으로 상, 하, 좌, 우 중 1이 있다면 이 부분에 간선이 있는 것으로 생각하면 그래프로 볼 수 있다.

문제에서 주어진 입력 예를 그래프로 나타내면 다음과 같다.

입력 예	대응되는 그래프
7 0 1 1 0 1 0 0 0 1 1 0 1 0 1 1 1 1 0 1 0 1 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0 1 1 1 1 1 0 0 1 1 1 0 0 0	

이와 같이 그래프로 만든 다음 배열의 (0, 0)부터 순차탐색을 진행하면서 (a, b)의 값이 만약 1이라면 이 점을 시작으로 하여 깊이우선탐색을 이용하여 모든 연결된 점을 방문하고 특정 값으로 체크한다.

나머지 점들도 모두 순차탐색하면서 마지막 까지 깊이우선탐색을 실행하고 알고리즘을 종료한다. 마지막에 깊이우선탐색을 실행한 횟수가 두더지의 수가 되고, 각 두더지 굴의 크기는 다른 배열에 저장해 둔 다음 마지막에 std::sort()를 이용하여 정렬한 후 내림차순으로 출력하면 된다.

여기에 사용되는 알고리즘은 지뢰찾기, 뿌요뿌요 등의 게임에 많이 활용되는 방법으로서, flood fill이라고도 한다. 자주 등장하는 방법이므로 익혀두면 활용가치가 크다.

이 알고리즘에서는 재귀함수를 이용하여 깊이우선탐색을 구현한다. 이때 가장 조심해야 할 점은 재귀의 깊이가 너무 커지면 runtime error가 발생할 수도 있다는 것이다. 일반적으로 release 모드라면 재귀의 깊이는 대략 10만 내외가 된다. 이 문제에서는 관계없지만 깊이가 너무 크다고 판단되면 다음 절에서 배울 너비우선탐색으로 처리하거나, 재귀 대신 스택을 이용해도 된다.

다음은 위 알고리즘의 실행 과정을 나타낸다.

0	1	1	0	1	0	0
0	1	1	0	1	0	1
1	1	1	0	1	0	1
0	0	0	0	1	1	1
0	1	0	0	0	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0

Size	0	0	0	0	0	0
index	0	1	2	3	4	5

[두더지 = 2]

두더지의 값은 2부터 시작 1은 주인 없는 굴이므로 2부터 증가함.

(0, 0)에서 탐색을 시작함.
(0, 0)의 원소가 0이므로 통과.

Size배열은 각 두더지 집의 크기를 저장할 배열

0	**1**	1	0	1	0	0
0	1	1	0	1	0	1
1	1	1	0	1	0	1
0	0	0	0	1	1	1
0	1	0	0	0	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0

Size	0	0	0	0	0	0
index	0	1	2	3	4	5

(0, 1)을 탐색, 원소의 값이 1이므로 dfs를 이용하여 상, 하, 좌, 우로 연결된 그래프를 모두 탐색하여 2로 수정함.

방문한 정점의 수인 7을 Size[2]에 기록하여 크기를 저장하고, 두더지 값 1 증가

0	2	**2**	0	1	0	0
0	2	2	0	1	0	1
2	2	2	0	1	0	1
0	0	0	0	1	1	1
0	1	0	0	0	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0

Size	0	0	7	0	0	0
index	0	1	2	3	4	5

[두더지 = 3]

(0, 2)를 탐색, 원소의 값이 원래 1이었으나 (0, 1)에 의해 2로 바뀌었으므로, 이미 다른 두더지의 굴에 포함되었음.

따라서 그냥 통과!

0	2	2	0	**1**	0	0
0	2	2	0	1	0	1
2	2	2	0	1	0	1
0	0	0	0	1	1	1
0	1	0	0	0	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0

Size	0	0	7	0	0	0
index	0	1	2	3	4	5

[두더지 = 3]

(0, 3), (0, 4)는 모두 패스, (0, 5)에서 다시 1이 등장하므로 이 점을 기준으로 dfs로 flood fill을 수행하면 상, 하, 좌, 우의 모든 칸들이 3으로 바뀜.

Size[3]을 방문한 정점의 수인 8로 채우고, 두더지의 값 1 증가

0	2	2	0	3	0	0
0	2	2	0	3	0	3
2	2	2	0	3	0	3
0	0	0	0	3	3	3
0	**1**	0	0	0	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0

Size	0	0	7	8	0	0
index	0	1	2	3	4	5

[두더지 = 4]

(0, 6)부터 (4, 0)까지는 1이 하나도 없으므로 모두 패스. (4, 1)에서 1이 등장하므로 이 칸으로부터 dfs로 모든 영역을 4로 채움.
그리고 방문한 정점의 수를 Size[4]에 기록함, 두더지 값은 5가 됨.

0	2	2	0	3	0	0
0	2	2	0	3	0	3
2	2	2	0	3	0	3
0	0	0	0	3	3	3
0	4	0	0	0	0	0
0	4	4	4	4	4	0
0	4	4	4	0	0	**0**

Size	0	0	7	8	9	0
index	0	1	2	3	4	5

[두더지 = 5]

마지막 칸까지 1의 값이 없으므로 모든 작업 종료.

세 마리 두더지가 있었고, 각 굴의 크기는 7, 8, 9임을 알 수 있음.

이 풀이에서는 특히 깊이우선탐색과 std::sort()를 내림차순 정렬하는 과정도 포함하고 있으므로, 잘 익혀두면 많은 도움이 될 것이다.

```
#include <stdio.h>
#include <algorithm>

int n, A[101][101], cnt, Size[101];

int main()
{
  input();
  solve();
  output();
}
```

기본적인 변수와 main 함수 부분이다. 입력, 풀이, 출력으로 따로 호출하고 있으며, 각 변수에 대한 설명은 다음과 같다.

배열 A는 전체 지도를 저장할 배열 (0은 땅, 1은 굴), 배열 Size는 각 두더지 굴의 크기를 저장할 배열, 배열 dx, dy는 현재 지점과 연결된 4곳의 x, y축 이동 양을 저장하는 배열이며, cnt는 총 두더지 굴의 수를 저장할 변수이다.

```
bool safe(int a, int b)
{
  return (0<=a && a<n) && (0<=b && b<n);
}
bool cmp(int a, int b)
{
  return a>b;
}
```

safe 함수는 이동해야할 장소가 지도의 경계를 넘었는지 검사하는 판정 함수, 지도를 벗어나는 곳이라면 false를 반환한다.

cmp는 정수를 기준으로 내림차순으로 정렬하기 위한 비교 함수

```
void dfs(int a, int b, int c)
{
  A[a][b]=c;
  if(safe(a+1, b) && A[a+1][b]==1)
    dfs(a+1, b, c);
  if(safe(a-1,b) && A[a-1][b]==1)
    dfs(a-1, b, c);
  if(safe(a,b+1) && A[a][b+1]==1)
    dfs(a, b+1, c);
  if(safe(a,b-1) && A[a][b-1]==1)
    dfs(a, b-1, c);
}
void solve()
{
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      if(A[i][j]==1)
      {
        cnt++;
        dfs(i,j,cnt+1);
      }
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      if(A[i][j])
        Size[A[i][j]-2]++;
  std::sort(Size, Size+cnt, cmp);
}
```

A배열의 (0, 0)부터 (n−1, n−1)까지 차례로 검사하면서 만약 굴의 일부가 발견되면, 그 부분으로부터 시작하여 dfs로 연결된 굴을 모두 검사한다.

dfs(a, b, c) : (a, b)의 정점과 연결된 모든 정점들을 c로 칠한다.

dfs 함수 부분의 4방향 탐색을 dx, dy를 이용하여 다음과 같이 편리하게 작성할 수 있다.

줄	코드	참고
1	`int dx[4]={1,0,-1,0}, dy[4]={0,1,0,-1};`	1: 4방향의 성분을 미리 설정한다.
2		
3	`void dfs(int a, int b, int c)`	
4	`{`	
5	`  A[a][b] = c;`	
6	`  for(int i=0; i<4; i++)`	
7	`    if(safe(a+dx[i],b+dy[i]) && A[a+dx[i]][b+dy[i]]==1)`	
8	`      dfs(a+dx[i], b+dy[i], c);`	
9	`}`	

이 방법은 앞으로도 이런 패턴의 문제에 다양하게 활용할 수 있으므로 활용법을 익힐 수 있도록 한다.

4방향으로 모두 탐색하며 탐색한 곳은 2이상의 값으로 바꾼다. 따라서 굴을 탐색하는 과정에서 1이 나오면 아직 확인하지 않은 두더지 굴이고 2이상의 값이 있다면 한 마리의 두더지의 굴로 확인했다는 의미로 해석할 수 있다.

22~25행은 각 굴의 크기를 Size배열에 채우는 과정을 나타낸다. 이 아이디어도 자주 활용하는 방법이므로 잘 익혀둘 수 있도록 한다. 26행은 Size의 내용을 내림차순으로 정렬하는 부분이다.

줄	코드	참고
1	`void input()`	
2	`{`	
3	`  scanf("%d", &n);`	
4	`  for(int i=0; i<n; i++)`	
5	`    for(int j=0; j<n; j++)`	
6	`      scanf("%d", &A[i][j]);`	
7	`}`	
8	`void output()`	
9	`{`	
10	`  printf("%d\n", cnt);`	
11	`  for(int i=0; i<cnt; i++)`	
12	`    printf("%d\n", Size[i]);`	
13	`}`	

각 값을 차례로 입력받는 input함수이다. 만약 입력 자료가 공백으로 구분되어 있지 않고 연속적으로 입력된다면 문자열 형태로 받을 수도 있지만 다음과 같이 처리할 수도 있다.

```
scanf("%1d",&A[i][j]);
```

위와 같이 입력받으면, 연속된 문자열로부터 1자씩 정수형으로 입력받는 것이 가능하다. 예를 들어 다음과 같이 주민등록 번호로부터 생년월일, 성별 등을 알고자 할 때, 다음과 같이 입력받으면 매우 편리하다.

```
scanf("%2d%2d%2d-%1d%d", &year, &mon, &day, &gender, &etc);
```

위와 같이 입력문을 사용하고 입력은 단순히 문자열 형태로 처리할 수 있다.

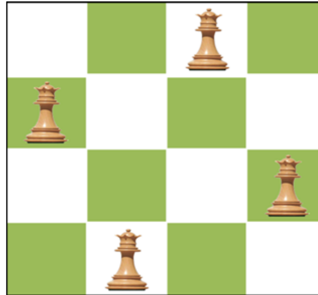
마지막으로 출력을 담당하는 함수인 output()은 먼저 굴의 수를 출력하고, 큰 굴부터 하나씩 출력한다.

문제 2

n-queen

전산학에서 백트래킹 문제로 n-queen problem이 유명하다.
이 문제는 n*n체스 보드판에 n개의 queen을 서로 공격하지 못하도록 배치하는 방법을 찾아내는 문제이다.

아래 그림은 n이 4일 경우 queen을 서로 공격하지 못하게 배치한 한 예를 나타낸다.



체스판 크기 및 queen의 수를 나타내는 n을 입력받아서 서로 공격하지 못하도록 배치하는 총 방법의 수를 구하는 프로그램을 작성하시오.

입력

정수 n이 입력으로 들어온다.(3 <= n <= 9)

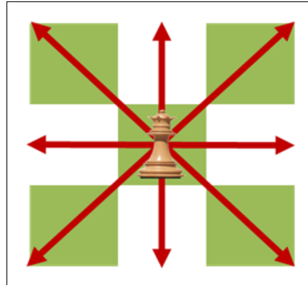
출력

서로 다른 총 경우의 수를 출력한다.

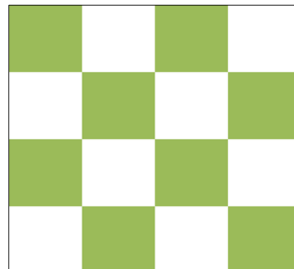
입력 예	출력 예
4	2

풀이

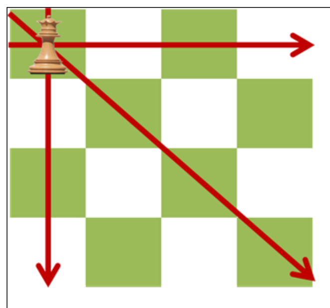
일단 이 문제를 풀기 위해서 퀸이 공격할 수 있는 위치에 대한 생각을 해야 한다. 일단 퀸이 공격할 수 있는 루트는 다음과 같다. (8방향으로 체스판의 마지막 칸까지 모두 공격 가능하다.)



이 문제를 해결하기 위하여 확실한 것은 한 행에 하나 이상의 퀸을 놓을 수 없다는 것이다. 4*4의 체스판을 살펴보자.



위 체스판에서 1행 1열에 하나의 퀸을 배치하면 공격범위는 아래 화살표와 같으며 화살표가 지나는 칸에는 퀸을 놓을 수 없다.



따라서 다음과 같은 방법을 활용할 수 있다.

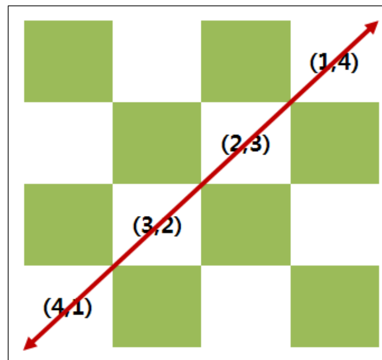
1. 첫 번째 행, 첫 번째 열에 퀸을 놓는다.
2. 다음 행에서 가능한 가장 왼쪽 열에 퀸을 놓는다.
3. n번째 열에 더 이상 퀸을 놓을 수 없다면 백트랙한다.
4. 마지막 행에 퀸을 놓으면 하나의 해를 구한 것이다.
5. 모든 경우를 조사할 때까지 백트래킹해가며 해들을 구한다.

위 방법으로 깊이우선탐색하며 해를 구할 때 마다 카운트하면 원하는 해를 구할 수 있다.

알고리즘 작성 시 주의할 점은 퀸을 놓을 수 있는지 없는지 판단하는 절차를 효율적으로 작성해야 한다.

이 풀이에서는 행은 검사할 필요가 없으므로, 열과 대각선만 검사하면 된다. 열을 검사하는 방법은 크기가 n인 체크배열을 만들어 k번째 열에 퀸을 놓았다면 배열의 k번째 위치를 체크한다. 체크하는 이유는 이후의 행에서는 체크된 열에 퀸을 놓지 않도록 하기 위함이다.

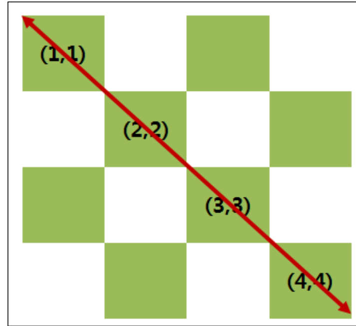
대각선은 기울기가 증가하는 대각선 부분과 기울기가 감소하는 부분의 2가지 대각선이 존재한다. 이 2가지 대각선에 대해서도 체크배열을 만들어서 활용할 수 있다. 기울기가 증가하는 대각선부터 살펴보면 다음과 같다.



위 대각선 상에 있는 칸의 특징을 보면 행+열의 값이 일정하다. n이 4일 경우 행+열의 최소값은 2이고 최댓값은 8이다.

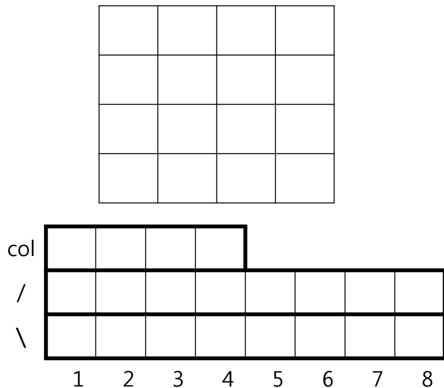
따라서 기울기가 증가하는 대각선은 체크배열의 행+열 위치에 체크하여 기울기가 증가하는 대각선 상에 퀸을 놓을 수 있는지 없는지를 쉽게 확인할 수 있다.

기울기가 감소하는 대각선도 아래와 같은 특징이 있다.



기울기가 감소하는 대각선 부분은 행과 열의 차가 일정하다. 범위는 n이 4일 경우 −3에서 3까지의 값을 지닌다. 음의 값을 양의 값으로 보정하기 위해 n을 더해주어 체크배열의 n+(행−열)의 위치에 체크하여, 퀸이 놓일 수 있는지 여부를 확인할 수 있다.

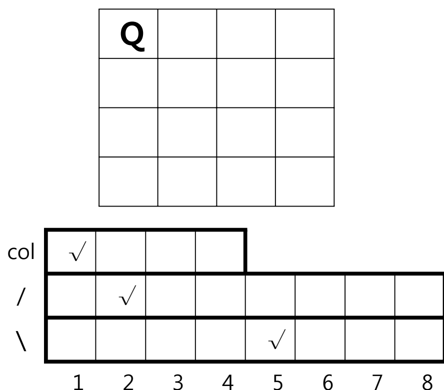
각 단계별 진행 과정은 다음과 같다.



[준비 상태]

사실 위에 있는 4*4의 체스판은 실제로 구현하지 않는다. 실제로 체스판으로 구현할 수도 있지만 이 방법보다는 여기서 소개하는 방법이 훨씬 효율적이며 속도도 빠르다.

col : 행, / : 대각선 1, \ : 대각선 2의 상태를 나타낸다.



[1단계]

1행 1열에 퀸을 하나 놓고,

col에 1열을 사용했기 때문에 col[1] 체크

대각선 1은 inc[1+1]에 체크

대각선 2는 dec[4-(1-1)]에 체크 (n=4이므로)

Q			
Q			

	1	2	3	4	5	6	7	8
col	√							
/		√						
\					√			

[2개 놓기]

2행 1열에 퀸을 놓아보자.
col[1]이 이미 체크되어 있으므로 놓을 수 없다.

Q			
	Q		

	1	2	3	4	5	6	7	8
col	√	√						
/		√		√				
\					√			

[2개 놓기]

다음으로 2행 2열에 퀸을 놓아보자.

col[2]는 체크 안 되었으므로 OK!
inc[2+2]도 체크 안 되어 있으므로 OK!
dec[4-(2+2)+1]가 이미 체크되었음. 즉 기울기가 감소하는 대각선에 퀸이 있다는 의미이므로 불가!

Q			
		Q	
Q	Q	Q	Q

	1	2	3	4	5	6	7	8
col	√		√					
/		√			√			
\				√	√			

[3개 놓기]

3행에는 1, 2, 3, 4열 모두 각각 체크 배열에 의해서 놓을 수 있는 위치가 없으므로 3행에는 퀸을 놓을 수 없다.

따라서 백트랙!!!

Q			

	1	2	3	4	5	6	7	8
col	√							
/		√						
\					√			

[백트랙]

백트랙 시에 가장 중요한 점은 체크배열에 기록해 두었던 체크를 모두 해제해야 한다는 점이다.

비선형구조의 탐색에서 복귀 시에 흔적을 지우는 것은 매우 중요한 요소이므로 익힐 수 있도록 한다.

Q			
			Q

	1	2	3	4	5	6	7	8
col	√			√				
/		√				√		
\			√		√			

[2개 놓기]

2행 3열까지는 아까 두었으므로, 2행 4열에 도전!!

col[4], inc[2+4], dec[4-(2-4)+1] 모두 비었으므로 둘 수 있음.

Q			
			Q
	Q		
Q	Q	Q	Q

	1	2	3	4	5	6	7	8
col	√	√		√				
/		√			√	√		
\			√		√	√		

[3개 놓기]

다음으로 3행 1열은 퀸을 놓을 수 없고, 3행 2열에 퀸을 놓을 수 있다.

마지막으로 4행에는 퀸을 놓을 수 있는 방법이 없으므로, 결국은 백트랙을 2번 하여 결국 1행 2열에 다시 놓게 된다.

	Q		

col		√						
/			√					
\				√				
	1	2	3	4	5	6	7	8

[1개 놓기]

1행 1열에 두면 가능한 방법이 없으므로, 다시 모두 백트랙한 후, 1행 2열에 놓고 다시 진행을 시작한다.

	Q		
			Q
Q			
		Q	

col	√	√	√	√				
/			√	√		√	√	
\			√	√		√	√	
	1	2	3	4	5	6	7	8

[계속 놓기]

다음으로 연속으로 깊이우선탐색을 진행하면 2행 4열, 3행 1열, 4행 3열에 각각 하나씩 퀸을 놓을 수 있고 한 가지의 가능한 경우를 찾을 수 있다.

다시 다른 해를 찾기 위해서 다시 백트랙 하여 계속 진행한다.

		Q	
Q			
			Q
	Q		

col	√	√	√	√				
/			√	√		√	√	
\			√	√		√	√	
	1	2	3	4	5	6	7	8

[계속 놓기]

마지막으로 1행 3열, 2행 1열, 3행 4열, 4행 2열로 또 다른 해를 찾을 수 있다.

따라서 모두 2가지의 서로 다른 경우를 발견할 수 있다.

이 방법을 종합하여 깊이우선탐색으로 해결한 소스코드는 다음과 같다.

줄	코드	참고
1	`#include<stdio.h>`	9: 마지막 행까지 다 놓았으면 해를 추가 10: 백트랙 12: r행에 대해서 각 열에 놓기 시도 15: 체크 17: 백트랙 후 흔적 제거(매우 중요)
2		
3	`int n, ans, col[10], inc[20], dec[20];`	
4		
5	`void solve(int r)`	
6	`{`	
7	`  if(r>n)`	
8	`  {`	
9	`    ans++;`	
10	`    return;`	
11	`  }`	
12	`  for(int i=1; i<=n; i++)`	
13	`    if(!col[i] && !inc[r+i] && !dec[n+(r-i)+1])`	
14	`    {`	
15	`      col[i]=inc[r+i]=dec[n+(r-i)+1]=1;`	
16	`      solve(r+1);`	
17	`      col[i]=inc[r+i]=dec[n+(r-i)+1]=0;`	
18	`    }`	
19	`}`	
20		
21	`int main()`	
22	`{`	
23	`  scanf("%d", &n);`	
24	`  solve(1);`	
25	`  printf("%d", ans);`	
26	`}`	

위 소스코드는 깊이우선탐색을 기반으로 퀸을 더 이상 못 놓는 상태라면 이전 상태로 백트랙하여 가능한 상태가 될 때까지 반복하는 것을 구현한 것이다.

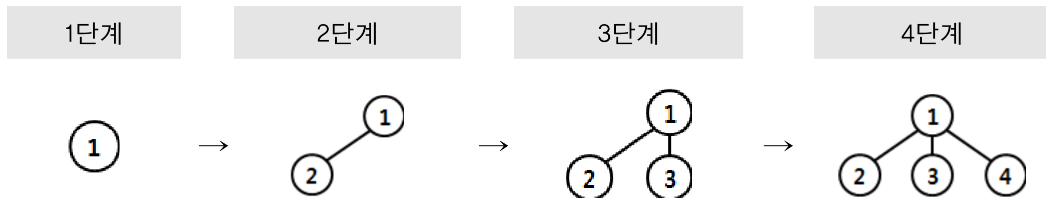
알고리즘의 효율을 높이기 위하여 퀸을 놓을 수 있는지 없는지를 $O(1)$ 만에 계산하기 위해, 현재 상태를 col, inc, dec라는 3개의 배열에 각각 열, 대각선 2가지의 상태를 저장하여 매우 빠른 속도로 처리할 수 있도록 하였다.

여기서 특별히 중요한 점은 다음 전체탐색을 위한 백트랙을 진행하면서 이전 전체탐색의 흔적을 지워야 한다는 것이다. 이 코드에서는 17행이 그 일을 하고 있다. 이 부분은 문

제의 특성에 따라 매우 중요할 수 있으므로 이 소스를 반드시 이해하여 활용할 수 있도록 해야 한다.

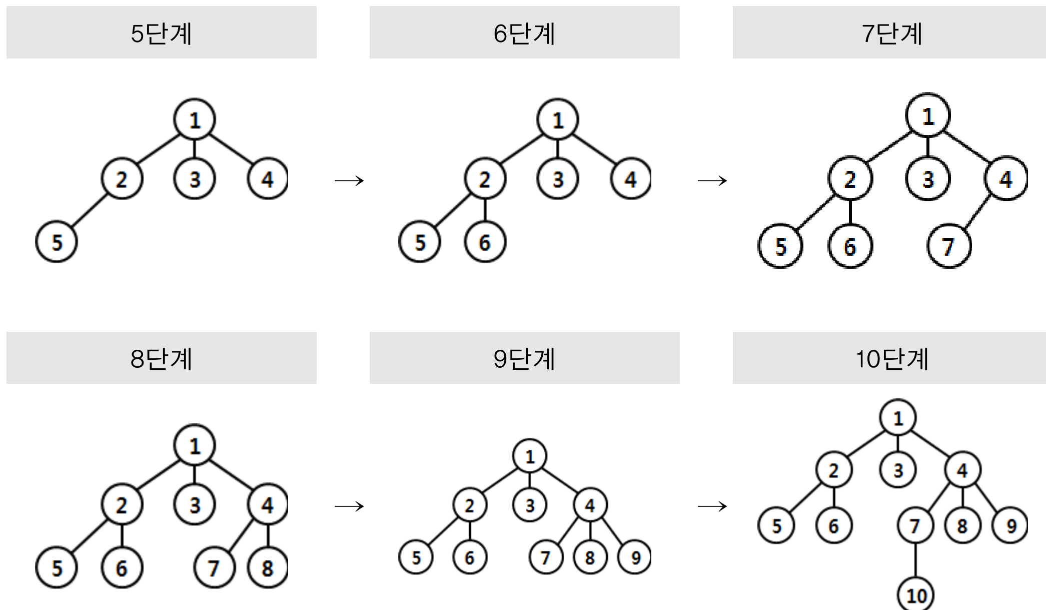
– 너비우선탐색(bfs)

너비우선탐색은 깊이우선탐색과는 달리 현재 정점에서 깊이가 1인 정점을 모두 탐색한 뒤 깊이를 늘려가는 방식이다. 73페이지의 '10개의 정점과 9개의 간선을 가진 트리'를 통해서 너비우선탐색을 살펴보자.



너비우선 1 ~ 4 단계

먼저 1단계부터 4단계까지를 살펴보면 1에서 출발하여 깊이가 1인 세 정점을 모두 순차적으로 방문한다. 계속해서 너비우선탐색의 결과를 살펴보면 다음과 같다.



너비우선 5 ~ 10 단계

너비우선탐색은 백트랙을 하지 않는다. 대신에 현재 정점에서 깊이가 1인 정점을 모두 방문해야 하므로 큐(queue)라는 선입선출(FIFO) 자료구조를 활용하여 현재 정점에서 깊이가 1 더 깊은 모든 정점을 순차적으로 큐에 저장하여 탐색에 활용한다. 따라서 STL에서 제공하는 std::queue()를 활용하는 방법을 익힐 필요가 있다.

너비우선탐색 알고리즘은 다음과 같다.

줄	코드	참고
1	`#include <queue>`	1: std::queue를 이용하기 위함
2	`bool visited[101];`	2: 방문했는지 체크해 두는 배열
3	`void bfs(int k)`	
4	`{`	
5	`  std::queue<int> Q;`	5: Queue를 선언
6	`  Q.push(k), visited[k]=1;`	6: 출발 정점을 Queue에 삽입
7	`  while(!Q.empty())`	7: Queue가 빌 때까지 반복
8	`  {`	
9	`    int current=Q.front(); Q.pop();`	9: Queue에서 하나 삭제
10	`    for(int i=0; i<G[current].size(); i++)`	10: 연결된 정점 모두 검사
11	`      if(!visited[G[current][i]])`	11: 아직 방문하지 않았으면,
12	`      {`	
13	`        visited[G[current][i]]=1;`	13: 체크 후 Queue에 추가
14	`        Q.push(G[current][i]);`	
15	`      }`	
16	`  }`	
17	`}`	

이 방법은 그래프를 인접리스트에 저장했을 경우에 활용할 수 있으며, 전체를 탐색하는 데 있어서 반복문의 실행횟수는 모두 m 번이 된다. 따라서 일반적으로 속도가 더 빠르기 때문에 자주 활용된다. 만약 인접행렬로 그래프를 저장했다면 다음과 같이 작성하면 된다.

하지만 표준 라이브러리(standard library)에 정의된 자료구조인 스택, 큐 등은 C++에서 쉽게 활용할 수 있지만 직접 구현하는 것보다 속도가 느리기 때문에 문제의 특성에 따라서 직접 구현하여 활용하는 것이 좋을 수도 있다.

줄	코드	참고
1	`#include <queue>`	1: std::queue()를 이용하기 위함
2	`bool visited[101];`	2: 방문했는지 체크해 두는 배열
3	`void bfs(int k)`	
4	`{`	
5	`  std::queue<int> Q;`	5: Queue를 선언
6	`  Q.push(k), visited[k]=1;`	6: 출발 정점을 Queue에 삽입
7	`  while(!Q.empty())`	7: Queue가 빌 때까지 반복
8	`  {`	
9	`    int current=Q.front(); Q.pop();`	9: Queue에서 하나 삭제
10	`    for(int i=1; i<=n; i++)`	10: 모든 정점에 대해 검사
11	`      if(G[current][i] && !visited[G[current][i]])`	11:검사하는 정점이 현재 정점과 연결되어 있고, 아직 방문하지 않았으면
12	`      {`	
13	`        visited[G[current][i]]=1;`	13: 체크 후 Queue에 추가
14	`        Q.push(G[current][i]);`	
15	`      }`	
16	`  }`	
17	`}`	

이 방법은 전체를 탐색하는 데 있어서 반복문을 n^2번 실행하게 된다. 따라서 평균적으로 인접리스트보다 느리지만 구현이 간편하므로, n값이 크지 않은 문제라면 충분히 적용할 가치가 있다.

문제 3

두더지 굴(L)

정올이는 땅속의 굴이 모두 연결되어 있으면 이 굴은 한 마리의 두더지가 사는 집이라는 사실을 발견하였다.

정올이는 뒷산에 사는 두더지가 모두 몇 마리인지 궁금해졌다. 정올이는 특수 장비를 이용하여 뒷산의 두더지 굴을 모두 나타낸 지도를 만들 수 있었다.

이 지도는 직사각형이고 가로 세로 영역을 0또는 1로 표현한다. 0은 땅이고 1은 두더지 굴을 나타낸다. 1이 상하좌우로 연결되어 있으면 한 마리의 두더지가 사는 집으로 정의할 수 있다.

0	1	1	0	1	0	0
0	1	1	0	1	0	1
1	1	1	0	1	0	1
0	0	0	0	1	1	1
0	1	0	0	0	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0

[그림 1]

0	1	1	0	2	0	0
0	1	1	0	2	0	2
1	1	1	0	2	0	2
0	0	0	0	2	2	2
0	3	0	0	0	0	0
0	3	3	3	3	3	0
0	3	3	3	0	0	0

[그림 2]

[그림 2]는 [그림 1]을 두더지 굴로 번호를 붙인 것이다. 특수촬영 사진 데이터를 입력받아 두더지 굴의 수를 출력하고, 각 두더지 굴의 크기를 오름차순으로 정렬하여 출력하는 프로그램을 작성하시오.

입력

첫 번째 줄에 가로, 세로의 크기를 나타내는 n이 입력된다. n은 30 이하의 자연수 두 번째 줄부터 n줄에 걸쳐서 n개의 0과 1이 공백으로 구분되어 입력된다.

출력

첫째 줄에 두더지 굴의 수를 출력한다. 둘째 줄부터 각 두더지 굴의 크기를 내림차순으로 한 줄에 하나씩 출력한다.

입력 예	출력 예
7 0 1 1 0 1 0 0 0 1 1 0 1 0 1 1 1 1 0 1 0 1 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0 1 1 1 1 1 0 0 1 1 1 0 0 0	3 9 8 7

풀이

이 문제는 깊이우선탐색으로 해결했던 문제이다. 하지만 너무 깊은 깊이에 대한 깊이우선탐색의 단점인 runtime error를 방지하기 위해서는 너비우선탐색을 적용할 수 있다. 이번 풀이는 너비우선탐색을 적용하여 이 문제를 해결한다.

기본적인 입력에 대한 그래프 처리 및 문제해결의 전반적인 방법은 앞의 문제를 참고하고, 연결된 정점을 처리하는 방법은 너비우선탐색을 이용한다.

이번 풀이에서는 flood fill을 너비우선탐색으로 처리하는 방법에 대해서 익혀보자. 소스코드는 다음과 같다.

코드	참고

```
#include <stdio.h>
#include <algorithm>
#include <queue>

struct VERTEX{ int a, b; };
int n, A[101][101], cnt, Size[101];
int dx[4]={1,0,-1,0}, dy[4]={0,1,0,-1};

int main()
{
  input();
  solve();
  output();
  return 0;
}
```

기본적인 변수와 main()함수 부분이다. 깊이우선탐색 때와의 차이점은 VERTEX라는 구조체를 선언한 부분과 queue를 삽입한 것이 차이가 난다. 이 부분은 queue를 활용하여 너비우선탐색을 하기 위하여 추가된 부분이다.

코드	참고

```
bool safe(int a, int b)
{
  return (0<=a && a<n) && (0<=b && b<n);
}
bool cmp(int a, int b)
{
  return a > b;
}
```

이 부분은 깊이우선탐색 때와 변함이 없다.

```
void bfs(int a, int b, int c)
{
  std::queue<VERTEX> Q;
  Q.push( (VERTEX){a, b} ), A[a][b]=c;
  while(!Q.empty())
  {
    VERTEX curr = Q.front(); Q.pop();
    for(int i=0; i<4; i++)
      if(safe(curr.a+dx[i], curr.b+dy[i]) &&
A[curr.a+dx[i]] [curr.b+dy[i]]==1)
      {
        A[curr.a+dx[i]][curr.b+dy[i]]=c;
        Q.push((VERTEX){curr.a+dx[i], curr.b+dy[i]});
      }
  }
}

void solve()
{
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      if(A[i][j]==1)
      {
        cnt++;
        bfs(i,j,cnt+1);
      }
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      if(A[i][j])
        Size[A[i][j]-2]++;
  std::sort(Size, Size+cnt, cmp);
}
```

bfs 함수의 핵심적인 부분이다. 일단 solve 함수에서 A배열의 (0, 0)부터 (n-1, n-1)까지 차례로 검사하면서 만약 굴의 일부가 발견되면, 그 부분으로부터 시작하여 bfs로 연결된 굴을 모두 검사한다.

bfs(a, b, c) : (a, b)의 정점과 연결된 모든 정점들을 c로 칠한다.

다른 부분은 모두 깊이우선탐색과 동일하나 bfs 함수의 내용을 잘 익혀둘 필요가 있다. 일단 시작정점을 큐에 삽입하고, 이 정점에서 4방향으로 연결된 모든 정점을 큐에 저장해 나간다. 이 때, 이미 큐에 들어있는 정점은 다시 넣지 않는다. 이 부분은 큐를 다루는 알고리즘에서 효율에 매우 큰 영향을 미치므로 반드시 익혀둘 수 있도록 한다.

큐에서 구조체를 이용하는 것도 활용도가 높으므로 구조체를 처리하는 부분의 코드들은 익혀두었다가 언제든지 활용할 수 있도록 연습하는 것이 중요하다.

```
void input()
{
  scanf("%d", &n);
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      scanf("%d", &A[i][j]);
}
void output()
{
  printf("%d\n", cnt);
  for(int i=0; i<cnt; i++)
    printf("%d\n", Size[i]);
}
```

각 값을 차례로 입력받는 input함수이다. 만약 입력 자료가 공백으로 구분되어 있지 않고 연속적으로 입력된다면 문자열 형태로 받을 수도 있지만 scanf("%1d",&A[i][j]) 로 입력받으면 처리할 수 있다.

출력하는 부분은 먼저 굴의 수를 출력하고, 크기가 큰 굴부터 하나씩 출력한다.

문제 4

미로 찾기

크기가 h*w인 미로가 있다.

이 미로는 길과 벽으로 구성되어 있으며, 길은 ".", 벽은 "#"으로 구성되어 있으며, 시작위치 "S"와 도착위치 "G"가 존재한다.

위에서 제시한 각 정보가 주어질 때, S위치로부터 G위치까지의 최단 거리를 구하는 프로그램을 작성하시오.

입력

첫 번째 줄에 h와 w가 공백으로 구분되어 입력된다.
(단, h, w는 5 이상 100 이하의 자연수이다.)
두 번째 줄부터 h줄에 걸쳐서 w개로 이루어진 문자열이 입력된다.
문자열은 길은 ".", 벽은 "#", 출발점은 "S", 도착점은 "G"로 표시된다. 그리고 S와 G의 위치는 서로 다르다

출력

출발지로부터 도착지까지의 최단거리를 출력한다.
단, 도달할 수 없는 미로일 경우에는 -1을 출력한다.

입력 예	출력 예
5 5 #S### #...# #.#.# #.... ###G#	 6

풀이

최단경로의 길이 즉, 최단거리를 찾는 문제는 너비우선탐색으로 해결할 수 있는 대표적인 예이다. 특히 이 문제의 경우에는 특별히 가중치 없이 이동하는 칸의 수가 최단거리 이므로 너비우선탐색을 적용하면 쉽게 해결할 수 있는 문제이다.

따라서 S로부터 출발하여 G까지 모두 6번의 이동으로 도착하는 것이 최소이다. 너비우선탐색은 출발정점에서 가까운 정점들로부터 탐색해나가기 때문에 도착정점까지의 최단거리를 더 쉽게 찾을 수 있다. 일단 주어진 예제를 이용하여 너비우선탐색을 진행해 나가는 과정을 살펴보면 다음과 같다.

```
#S###
#...#
#.#.#
#....
###G#
```

S의 위치 (0, 1)을 먼저 큐에 넣고 탐색을 시작한다.

미로는 현재 원래의 입력과 다름없다.

Queue							
	0,1						

```
#S###
#1..#
#.#.#
#....
###G#
```

큐에서 자료를 하나 뺀다. 뺀 좌표가 (0, 1)이므로 이 좌표와 상하좌우에 위치한 칸들 중 이동가능한 모든 칸은 맵 상에 1을 기록하고 큐에 넣는다.

(1,1)만 이동 가능하므로 (1,1)만 큐에 삽입된다.

Queue							
	1,1						

```
#S###
#12.#
#2#.#
#....
###G#
```

큐에서 자료를 하나 뺀다. 삭제된 좌표가 (1, 1)이다.
이 좌표와 상하좌우로 인접한 좌표 중 아직 방문하지 않았으면서 이동가능한 모든 위치의 맵의 (1, 1) 위치의 값 + 1을 기록하고, 모두 큐에 삽입한다.
이때는 (1, 2)와 (2, 1)이 삽입된다.

Queue							
	1,2	2,1					

```
#S###
#123#
#2#.#
#....
###G#
```

이번에 큐에서 빠진 좌표는 (1, 2)이다. 따라서 여기서 이동가능한 곳은 (1, 3) 뿐이므로 이곳에 3이 기록되고 큐에 입력된다.

Queue							
	2,1	1,3					

```
#S###
#123#
#2#.#
#3...
###G#
```

다음으로 큐에서 (2, 1)을 삭제하고, 이 좌표에서 이동 가능한 좌표인 (3, 1)에 3을 기록하고 다시 큐에 삽입한다.

Queue							
	1,3	3,1					

```
#S###
#123#
#2#4#
#3...
###G#
```

다음으로 (1, 3)이 큐에서 제거되고, 제거된 좌표에서 이동 가능한 (2, 3)에 4를 기록하고 큐에 삽입

Queue							
	3,1	2,3					

```
#S###
#123#
#2#4#
#34..
###G#
```

다음으로 (3, 1)이 큐에서 제거되고, 제거된 좌표로부터 이동 가능한 (3, 2)에 4를 기록하고, 큐에 삽입

Queue							
	2,3	3,2					

```
#S###
#123#
#2#4#
#345.
###G#
```

다음으로 (2, 3)이 큐에서 제거되고, 제거된 좌표로부터 이동 가능한 (3, 3)에 5를 기록하고, 큐에 삽입

Queue							
	3,2	3,3					

```
#S###
#123#
#2#4#
#345.
###G#
```

다음으로 (3, 2)가 큐에서 제거되고, 제거된 좌표로부터 아직 방문하지 않았거나 이동 가능한 정점이 없으므로 그냥 패스!

Queue							
	3,3						

```
#S###
#123#
#2#4#
#3456
###6#
```

다음으로 (3, 3)이 큐에서 제거되고, 제거된 좌표로부터 이동 가능한 (3, 4)와 (4, 3)을 모두 큐에 삽입함.

Queue							
	3,4	4,3					

```
#S###
#123#
#2#4#
#3456
###6#
```

큐에서 (3, 4)를 제거하고 이 좌표로부터 더 이상 이동 가능한 좌표가 없으므로, 다시 큐에서 (4, 3)을 제거한다.

(4, 3)은 목표지점의 좌표이므로, 더 이상 탐색을 진행할 필요가 없다. 알고리즘은 종료되고, 출발지로부터 목적지까지의 최단길이는 6임을 알 수 있다.

Queue							

이 문제를 풀 때, 입력 자료를 문자열의 형태로 받아야하므로 주의해야 하며 큐를 이용하여 너비우선탐색을 구현하는 방법으로 해결해보자.

```
#include <stdio.h>
#include <queue>

struct VERTEX{ int a, b; };
int h, w, Sa, Sb, Ga, Gb, visited[101][101];
int dx[4]={1,0,-1,0}, dy[4]={0,1,0,-1};
char M[101][101];

bool safe(int a, int b)
{
  return (0<=a && a<h) && (0<=b && b<w);
}

int main()
{
  input();
  printf("%d\n", solve());
}
```

각 변수 h, w는 전체 미로의 높이와 폭을 가지는 변수이고, Sa, Sb는 출발점의 좌표, Ga, Gb는 도착점의 좌표이며, visited는 각 정점까지의 거리를 기록하면서 현재 정점을 방문한지 안한지를 체크하는 용도로 사용되며, dx, dy는 이동 가능한 4방향을 설정한다.

구조체 VERTEX는 미로의 한 칸을 나타내는 구조체로 좌표값을 가진다. M은 미로의 각 칸이 어떤 값으로 구성되었는지 저장하는 배열로 활용된다.

safe는 이동하려고 하는 정점이 실제 미로의 내부인지 아닌지 판단하는 역할을 하는 함수이다.

줄	코드	참고

```
void input(void)
{
  scanf("%d %d", &h, &w);
  for(int i=0; i<h; i++)
  {
    scanf("%s", M[i]);
    for(int j=0; j<w; j++)
      if(M[i][j]=='S') Sa=i, Sb=j;
      else if(M[i][j]=='G') Ga=i, Gb=j, M[i][j]='.';
  }
}

int solve(void)
{
  std::queue<VERTEX> Q;
  Q.push((VERTEX){Sa, Sb}), visited[Sa][Sb] = 0;
  while(!Q.empty())
  {
    VERTEX cur=Q.front(); Q.pop();
    if(cur.a==Ga && cur.b==Gb) break;

      for(int i=0; i<4; i++)
      {
        int a=cur.a+dx[i], b=cur.b+dy[i];
        if(safe(a, b) && !visited[a][b] && M[a][b]=='.')
        {
          visited[a][b]=visited[cur.a][cur.b]+1;
          Q.push((VERTEX){a, b});
        }
      }
  }
  return visited[Ga][Gb];
}
```

위 소스코드는 핵심적인 부분이다. 먼저 입력부에서 중요한 점은 도착점의 값을 'G'에서 '.'로 바꾼다. 마지막 도착점 또한 이동 가능한 상태로 두어야 더 쉬운 코딩이 가능하기 때문이며, 도착 여부의 판단은 좌표를 이용하면 된다.

28행에서 구조체에 자료를 입력하는 부분의 코드가 익숙하지 않을 수 있다. 일반적으로는 28행의 내용을 처리하는 코드는 다음과 같다.

줄	코드	참고
1	`if(safe(a, b) && !visited[a][b] && M[a][b]=='.')`	
2	`{`	
3	`  VERTEX temp;`	
4	`  temp.a=a;`	
5	`  temp.b=b;`	
6	`  visited[a][b]=visited[cur.a][cur.b]+1;`	
7	`  Q.push(temp);`	
8	`}`	

하지만 구조체에 값을 원소나열법과 같이 순서대로 나열하고 “{ }” 로 묶어서 대입하면 구조체로 처리할 수 있다. 그리고 형 변환을 해주면 보다 확실하게 처리할 수 있다. 따라서 다음과 같은 코드로 변경 가능하다.

줄	코드	참고
1	`if(safe(a, b) && !visited[a][b] && M[a][b]=='.' )`	
2	`{`	
3	`  VERTEX temp=(VERTEX){ a, b };`	
4	`  visited[a][b]=visited[cur.a][cur.b]+1;`	
5	`  Q.push(temp);`	
6	`}`	

마지막을 VERTEX의 선언 없이 직접 대입으로 28행과 같이 처리할 수 있다.

줄	코드	참고
1	`if(safe(a, b) && !visited[a][b] && M[a][b]=='.')`	
2	`{`	
3	`  visited[a][b]=visited[cur.a][cur.b]+1;`	
4	`  Q.push((VERTEX){ a, b });`	
5	`}`	

다음으로 13행부터 31행까지는 너비우선탐색을 구현한 부분이다. 먼저 시작점을 큐에 삽입하고, 큐가 빌 때까지 아직까지 방문하지 않은 정점들을 차례로 큐에 삽입한다. 큐의 특성 상, 출발점에서 가까운 정점들이 먼저 큐에 삽입된다.

여기서 중요한 점은 visited라는 배열에는 출발점과의 거리가 기록되도록 코딩한다는 점

이다. 방문했으면 1, 아니면 0으로 기록할 수도 있지만, 이 문제의 경우 방문하지 않았으면 0, 방문했으면 이동거리를 저장하는 아이디어를 이용하여 문제를 해결하고 있다.

이와 같이 다양한 아이디어를 이용하여 문제를 해결할 수 있기 때문에 평소에 다양한 관점에서 문제를 접근하는 연습을 한다면 창의적인 문제해결력이 향상된다.

5 전체탐색법

전체탐색법은 모든 문제해결의 기초가 되는 가장 중요한 설계법 중 하나라고 할 수 있다. 주어진 문제에서 해가 될 수 있는 모든 가능성을 검사하여 해를 구하기 때문에 항상 정확한 해를 구할 수 있다는 점이 장점이다. 하지만 탐색해야할 내용이 너무 많으면 문제에서 제시한 시간 이내에 해결할 수 없다는 점을 유의해야 한다.

하지만 전체탐색을 기반으로 한 다양한 응용들이 있으며, 이러한 응용들을 통하여 탐색해야할 공간을 배제해 나가면서 시간을 줄일 수 있는 다양한 방법들이 존재하기 때문에 잘 응용하면 많은 문제를 해결할 수 있는 강력한 도구가 될 수 있다. 따라서 전체탐색법을 잘 익혀두면 다른 알고리즘 설계법을 학습하는데 많은 도움이 된다.

전체탐색법은 앞 단원들에서 공부한 선형구조의 탐색, 비선형구조의 탐색을 기반으로 하여 문제를 해결한다.

가. 선형구조와 비선형구조의 전체탐색

선형구조의 전체탐색은 앞에서 배운 대로 주로 반복문을 이용하여 접근할 수 있다. 1차원 뿐만 아니라 2차원 이상의 다차원 구조에 대해서도 선형구조로 탐색할 수 있다.

비선형구조의 전체탐색은 문제해결의 가장 기본이 되는 알고리즘 설계법인 백트래킹이다. 백트래킹 기법은 재귀함수를 이용하여 간단하게 구현할 수 있고, 다양한 문제를 해결하는데 많이 응용되는 방법이므로 반드시 익혀둘 필요가 있다.

주어진 문제들을 통하여 선형구조, 비선형구조의 전체탐색법을 익힐 수 있도록 하자.

문제 1

약수의 합 구하기 1

한 정수 n을 입력받아서 n의 모든 약수의 합을 구하는 프로그램을 작성하시오.

예를 들어 10의 약수는 1, 2, 5, 10이므로 이 값들의 합인 18이 10의 약수의 합이 된다.

입력

첫 번째 줄에 정수 n이 입력된다.
(단, 1 <= n <= 100,000)

출력

n의 약수의 합을 출력한다.

입력 예	출력 예
10	18

풀이

이 문제는 기본적으로 수학적인 아이디어를 이용하여 해결할 수 있는 문제이지만 이 단원에서는 전체탐색법을 다루는 단원이므로 전체탐색법으로 해결해보자.

일단 n을 입력받으면 1부터 n까지의 모든 수를 차례로 반복문을 이용하여 선형으로 탐색하면서 n의 약수들을 검사한다. 만약 현재 탐색 중인 수가 n의 약수라면 누적하여 구할 수 있다. 이렇게 구한다면 계산량은 $O(n)$이 된다. 이 문제에서는 n의 최댓값이 100,000이므로 충분히 해결할 수 있는 문제가 된다.

어떤 수 x가 n의 약수라면 다음 조건을 이용해 구할 수 있다.

```
n % x == 0
```

이를 이용하여 문제를 해결한 소스코드는 다음과 같다.

```c
#include <stdio.h>

int n;

int solve()
{
  int ans = 0;
  for(int i=1; i<=n; i++ )
    if(n%i==0)
      ans+=i;
  return ans;
}

int main()
{
  scanf("%d", &n);
  printf("%d\n", solve());
}
```

이 문제는 이와 같은 방법으로 쉽게 해결할 수 있으나, n이 10억 이상의 값으로 커질 때는 다른 방법을 생각해야 한다. 나중에 다루게 될 것이므로 한 번 생각해보자.

문제 2

최댓값(L)

<그림 1>과 같이 9×9 격자판에 쓰여진 81개의 자연수가 주어질 때, 이들 중 최댓값을 찾고 그 최댓값이 몇 행 몇 열에 위치한 수인지 구하는 프로그램을 작성하시오.

예를 들어, 다음과 같이 81개의 수가 주어질 경우에는 이들 중 최댓값은 90이고, 이 값은 5행 7열에 위치한다.

	1열	2열	3열	4열	5열	6열	7열	8열	9열
1행	3	23	85	34	17	74	25	52	65
2행	10	7	39	42	88	52	14	72	63
3행	87	42	18	78	53	45	18	84	53
4행	34	28	64	85	12	16	75	36	55
5행	21	77	45	35	28	75	90	76	1
6행	25	87	65	15	28	11	37	28	74
7행	65	27	75	41	7	89	78	64	39
8행	47	47	70	45	23	65	3	41	44
9행	87	13	82	38	31	12	29	29	80

<그림 1>

입력

첫째 줄부터 아홉째 줄까지 한 줄에 아홉 개씩 자연수가 주어진다. 주어지는 자연수는 100보다 작다.

출력

첫째 줄에 최댓값을 출력하고, 둘째 줄에 최댓값이 위치한 행 번호와 열 번호를 빈칸을 사이에 두고 차례로 출력한다. 최댓값이 두 개 이상인 경우 그 중 한 곳의 위치를 출력한다.

입력 예

```
3 23 85 34 17 74 25 52 65
10 7 39 42 88 52 14 72 63
87 42 18 78 53 45 18 84 53
34 28 64 85 12 16 75 36 55
21 77 45 35 28 75 90 76 1
25 87 65 15 28 11 37 28 74
65 27 75 41 7 89 78 64 39
47 47 70 45 23 65 3 41 44
87 13 82 38 31 12 29 29 80
```

출력 예

```
90
5 7
```

출처: 한국정보올림피아드(2007 지역예선 중고등부)

풀이

이 문제는 2차원 구조를 선형으로 모두 탐색하면 쉽게 해결할 수 있는 문제이다. 2차원 구조는 행 우선으로 탐색하는 방법과 열 우선으로 탐색하는 방법이 있는데, 이 문제는 어떤 방법으로 탐색해도 관계없으며, 일반적으로는 행 우선 탐색을 많이 사용한다.

5행 4열의 2차원 배열

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20

[2차원 구조에서의 행 우선 탐색 순서]

5행 4열의 2차원 배열

1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19
5	10	15	20

[2차원 구조에서의 열 우선 탐색 순서]

다음은 행 우선을 반복문으로 구현한 소스코드이다.

```
for(int row=0; row<5; row++)
{
  for(int col=0; col<4; col++)
    printf("[%d, %d]", row, col);
  puts("");
}
```

다음은 열 우선을 반복문으로 구현한 소스코드이다.

```
for(int col=0; col<4; col++)
{
  for(int row=0; row<5 ; row++)
    printf("[%d, %d]\n", row, col);
  puts("");
}
```

[2차원 구조에서의 행 우선 출력 결과]	[2차원 구조에서의 열 우선 출력 결과]
[0, 0] [0, 1] [0, 2] [0, 3] [1, 0] [1, 1] [1, 2] [1, 3] [2, 0] [2, 1] [2, 2] [2, 3] [3, 0] [3, 1] [3, 2] [3, 3] [4, 0] [4, 1] [4, 2] [4, 3]	[0, 0] [1, 0] [2, 0] [3, 0] [4, 0] [0, 1] [1, 1] [2, 1] [3, 1] [4, 1] [0, 2] [1, 2] [2, 2] [3, 2] [4, 2] [0, 3] [1, 3] [2, 3] [3, 3] [4, 3]

이제 문제를 해결하는 방법에 대해서 알아보자.

탐색하기 전 먼저 해를 저장할 변수인 ans를 0으로 초기화한다. 여기서 주의할 점은 각 원소들 중 음수값이 존재할 경우 최댓값을 구하기 위해 ans를 0으로 초기화하면 안 된다는 점이다. 이 문제는 음수값이 존재하지 않기 때문에 ans를 0으로 초기화하고 문제를 해결한다.

참고로 어떤 변수에 값을 초기화하는 몇 가지 방법을 소개한다. 일단 int형의 최댓값은 0x7fffffff(2,147,483,647)이며, 최솟값은 0x80000000(-2,147,483,648)이다. 엄밀하게 최대, 최소를 지정할 때 이 값을 이용하면 되며, 16진법을 이용하면 쉽게 처리할 수 있다.

여기서 주의할 점은 위 값들을 설정한 후 값을 증가시키거나 감소시키면 오버플로(overflow)로 인하여 답이 잘못될 수 있다. 예를 들어 다음 명령을 보자.

줄	코드	참고
1	int max = 0x7fffffff;	
2	max = max + 1;	

위 예의 경우에 max값이 최댓값이었는데, 여기서 1을 증가하면 오버플로가 발생하여 max값은 음수가 된다. 따라서 이런 점을 방지하기 위하여 적어도 2배 정도라 하더라도 오버플로가 발생하지 않도록 처리하는 경우가 많다. 이럴 때는 주로 최댓값을 987654321 등의 자릿수도 쉽게 알 수 있고 2배를 하더라도 정수 범위에 있는 수 등을 활용하는 경우가 많다. 문제에 따라서는 탐색하고자 하는 데이터 중에서 임의의 한 값을 최댓값 또는 최솟값으로 결정하는 방법도 있다.

이러한 점들도 자신만의 코딩 스타일을 구성하는 요소가 되므로 자신만의 방식으로 최대, 최소 등을 정하는 방법을 익혀두자. 위 문제를 해결하는 소스코드는 다음과 같다.

```
#include <stdio.h>
int A[10][10], ans, mi, mj;
void input()
{
  for(int i=0; i<9; i++)
    for(int j=0; j<9; j++)
      scanf("%d", &A[i][j]);
}

int solve()
{
  for(int i=0; i<9; i++)
    for(int j=0; j<9 ; j++)
      if(ans < A[i][j])
      {
        ans=A[i][j];
        mi=i+1;
        mj=j+1;
      }
}

int main()
{
  input();
  solve();
  printf("%d\n%d %d\n", ans, mi, mj);
  return 0;
}
```

가장 일반적으로 해결할 수 있는 방법이고 이 경우 계산량은 O(행×열)이 된다. 이를 보다 효율적으로 바꾸기 위해서, 입력받으면서 바로 처리할 수도 있으며, ans, mi, mj를 모두 쓰지 않고 mi, mj만 가지고 처리하는 방법을 소개한다.

```c
#include <stdio.h>

int A[10][10], mi, mj;

void input_solve()
{
  for(int i=0; i<9; i++)
    for(int j=0; j<9; j++)
    {
      scanf("%d", &A[i][j]);
      if(A[mi][mj]<A[i][j])
        mi=i, mj=j;
    }
}

int main()
{
  input_solve();
  printf("%d\n%d %d\n", A[mi][mj], mi+1, mj+1);
  return 0;
}
```

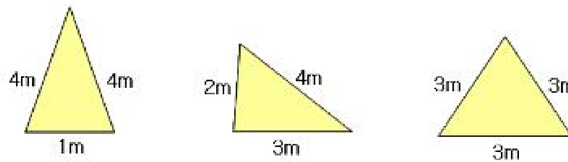
이 방법도 잘 이해하고 익혀두면 코딩의 실수와 시간을 줄일 수 있다.

문제 3

삼각화단 만들기(S)

주어진 화단 둘레의 길이를 이용하여 삼각형 모양의 화단을 만들려고 한다. 이 때 만들어진 삼각형 화단 둘레의 길이는 반드시 주어진 화단 둘레의 길이와 같아야 한다. 또한, 화단 둘레의 길이와 각 변의 길이는 자연수이다. 예를 들어, 만들고자 하는 화단 둘레의 길이가 9m라고 하면,

- 한 변의 길이가 1m, 두 변의 길이가 4m인 화단
- 한 변의 길이가 2m, 다른 변의 길이가 3m, 나머지 변의 길이가 4m인 화단
- 세 변의 길이가 모두 3m인 3가지 경우의 화단을 만들 수 있다.



화단 둘레의 길이를 입력받아서 만들 수 있는 서로 다른 화단의 수를 구하는 프로그램을 작성하시오.

입력

화단의 길이 n이 주어진다.(단, 1 <= n <= 100)

출력

출력내용은 입력받은 n으로 만들 수 있는 서로 다른 화단의 수를 출력한다.

입력 예	출력 예
9	3

출처: 한국정보올림피아드(2002 전국본선 초등부)

풀이

이 문제는 입력구조로 볼 때, 전체탐색으로 해결하려면 선형구조인지 비선형구조인지 등을 판단하기 쉽지 않다. 즉, 지금까지 다루었던 문제들보다 문제를 구조화하는 데에 조금 더 어려움이 있는 문제라고 할 수 있다.

이 문제에서는 세 변의 길이의 합인 n을 알고 있는 상태에서 삼각형의 세 변의 길이를 구하는 문제이므로 각 변을 a, b, c라고 하면 변의 길이 a의 길이를 1부터 n까지 정하고, b, c도 같은 방법으로 순차적으로 정해나가는 방법으로 전체탐색을 할 수 있다. 이렇게 정할 경우 3차원 구조를 가지는 선형 구조가 된다.

여기서 주의할 점은 a, b, c를 각각 1부터 n까지 탐색한다면 각 삼각형이 여러 번 중복되어 구해진다. 예를 들어 화단의 길이가 9일 때, 2, 4, 3으로 골랐다면 3, 4, 2와 4, 2, 3등은 모두 같은 삼각형이지만 따로 카운팅하게 된다. 따라서 처음부터 a를 가장 짧은 변, c를 가장 긴 변으로 정하면 문제 해결이 간단해진다.

그리고 이 문제의 입력 n의 최댓값이 100이므로 100^3으로 접근하더라도 충분히 해결가능하기 때문에 전체탐색법으로 해결해보자.

먼저 3차원 구조로 세변의 길이를 전체 탐색하는 구문을 작성해보자.

코드	참고

```
int count=0;
for(int a=1; a<=n; a++)
  for(int b=a; b<=n; b++)
    for(int c=b; c<=n; c++)
    {
      if(count%5 == 0) puts("");
        count++;
      printf("[%d %d %d]\t", a, b, c);
    }
```

위 구조대로 탐색하면 각 변의 길이가 1부터 5까지의 모든 경우에 대해서 조사한다. 단 각 변의 길이를 a, b, c라고 할 때, $a \le b \le c$ 를 만족하는 값들만 탐색한다. 위 탐색의 결과를 출력하면 다음과 같다. 출력할 때, 한 줄에 5개씩만 출력하도록 count변수를 활용

하였으므로 참고한다.

[1 1 1]	[1 1 2]	[1 1 3]	[1 1 4]	[1 1 5]
[1 2 2]	[1 2 3]	[1 2 4]	[1 2 5]	[1 3 3]
[1 3 4]	[1 3 5]	[1 4 4]	[1 4 5]	[1 5 5]
[2 2 2]	[2 2 3]	[2 2 4]	[2 2 5]	[2 3 3]
[2 3 4]	[2 3 5]	[2 4 4]	[2 4 5]	[2 5 5]
[3 3 3]	[3 3 4]	[3 3 5]	[3 4 4]	[3 4 5]
[3 5 5]	[4 4 4]	[4 4 5]	[4 5 5]	[5 5 5]

위와 같이 탐색을 하면 모든 경우에 대해서 탐색한다. 따라서 각 건에 대해서 삼각형 여부만 판단하면 된다. 세 변의 길이로 삼각형을 판단하는 기본 조건은 다음과 같다.

$$a + b > c$$

그리고 이 문제에서만 적용되는 조건이 있다. 세 변의 길이의 합이 n이어야 한다. 따라서 다음 조건도 만족해야 한다.

$$a + b + c = n$$

위 탐색방법과 삼각형의 조건을 적용한 소스코드는 다음과 같다.

코드	참고

```
#include <stdio.h>

int n;

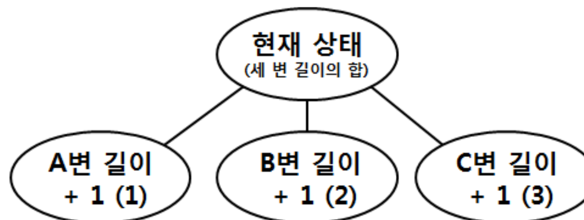
int solve()
{
  int cnt = 0;
  scanf("%d", &n);
  for(int a=1; a<=n; a++)
    for(int b=a; b<=n; b++)
      for(int c=b; c<=n; c++ )
        if(a+b+c == n && a+b>c)
          cnt++;
  return cnt;
}

int main()
{
  printf("%d\n", solve());
}
```

9행~13행이 3차원으로 공간을 탐색해 나가는 과정이다. 이러한 문제와 같이 직접적으로 구조화되어 있지 않은 문제도 해결과정에서 구조화해가며 풀어야 하는 문제가 많다.

12행에서는 세 변의 길이의 합 조건과 삼각형을 이루는 조건을 검사한다.

이 문제는 비선형으로 구조화해서 해결할 수도 있다. 문제에서 주어진 조건들을 이용하여 다음과 같은 탐색구조를 설계할 수 있다.



위 트리를 살펴보면 현재 상태(현재 세 변 길이의 합)가 이 n보다 적으면 (1), (2), (3)의 순으로 깊이우선탐색한다. 그러면 다시 (1), (2), (3)이 각각 새로운 현재 상태가 된다. 계속 깊이우선으로 탐색할 수 있는 상태가 된다.

만약 현재 상태가 n이 되면 탐색을 종료하고 삼각형의 조건이 맞는지 확인한다. a, b, c 세 변이 삼각형을 이루는 조건을 만족한다면 가능한 삼각형의 수를 1증가시킨다.

이렇게 모든 상태를 탐색하면 가능한 모든 경우가 만들어진다. 이를 구현한 소스코드는 다음과 같다.

줄	코드	참고
1	#include<stdio.h>	6: 모든 변을 다 썼으면 12: a변을 1증가 계속 탐색 13: b변을 1증가 계속 탐색 14: c변을 1증가 계속 탐색
2		
3	int cnt;	
4	void solve(int n, int a, int b, int c)	
5	{	
6	if(a+b+c==n)	
7	{	
8	if(a<=b && b<=c && a+b>c)	
9	cnt++;	

줄	코드	참고
10	return;	
11	}	
12	solve(n, a+1, b, c);	
13	solve(n, a, b+1, c);	
14	solve(n, a, b, c+1);	
15	}	
16		
17	int main(void)	
18	{	
19	int n;	
20	scanf("%d", &n);	
21	solve(n, 1, 1, 1);	
22	printf("%d\n", cnt);	
23	}	

하지만 위 소스코드에는 문제점이 있다. 각 변에 1씩 증가시켜가며 탐색해 나가는데 이는 n이 5이고 최종 상태가 1, 2, 2라고 할 때, b변에서 1을 먼저 증가시켜 나온 1, 2, 2와 c변에서 1을 먼저 증가시켜 나온 1, 2, 2를 서로 다른 경우로 카운트한다. 즉, 같은 모양의 삼각형을 여러 번 중복해서 계산하게 된다.

이를 방지하기 위하여 한 번 삼각형으로 카운트 된 a, b, c에 대해서는 chk[a][b][c]배열의 값을 1로 바꾸어 체크해둔다. 이 방법을 이용하여 카운트 하는 것을 방지할 수 있다.

위 방식으로 작성한 소스 프로그램은 아래와 같다.

줄	코드	참고
1	#include<stdio.h>	3: chk는 중복 체크용 7: 모든 변을 다 썼으면 9: 삼각형 조건 만족 12: 중복 방지용 체크 16: a변을 1증가 계속 탐색 17: b변을 1증가 계속 탐색 18: c변을 1증가 계속 탐색
2		
3	int cnt, chk[21][21][21];	
4		
5	void solve(int n, int a, int b, int c)	
6	{	
7	if(a+b+c==n)	
8	{	
9	if(a<=b && b<=c && a+b>c && chk[a][b][c]==0)	
10	{	
11	cnt++;	
12	chk[a][b][c]=1;	
13	}	

```c
    return;
  }
  solve(n, a+1, b, c);
  solve(n, a, b+1, c);
  solve(n, a, b, c+1);
}

int main(void)
{
  int n;
  scanf("%d", &n);
  solve(n, 1, 1, 1);
  printf("%d\n", cnt);
}
```

이와 같이 중복 방지를 위해서 체크하는 방법은 자주 활용되므로 잘 익혀둘 수 있도록 한다. 그리고 만약 n의 크기가 10,000이상의 값이라면 어떻게 해결해야할지 고민해보기 바란다. 이 방법들은 모두 시간이 너무 많이 걸리기 때문에 n값이 커질 경우 제한된 시간 이내에 해를 구할 수 없다.

문제 4

고기잡이(S)

우리나라 최고의 어부 정올이가 이번에 네모네모 배 고기잡이 대회에 참가한다.

이 대회에는 3개의 라운드가 있는데, 첫 번째 라운드는 1차원 형태로 표현될 수 있는 작은 연못에서 길쭉한 그물을 던져서 최대한 많은 고기를 잡는 것이 목적이다.

1라운드의 예를 들면 연못의 크기가 1*6이고 물고기의 위치와 가치가 다음과 같다고 하자.

1 0 2 0 4 3

여기서 그물의 크기는 1*3이라고 할 때, 잡을 수 있는 방법은 (1 0 2), (0 2 0), (2 0 4), (0 4 3)의 4가지 방법이 있다.

이 중 가장 이득을 보는 방법은 마지막 방법 0 + 4 + 3 = 7이다. 따라서 주어진 경우의 최대 이득은 7이 된다. 정올이는 최대한 가치가 큰 물고기를 잡아서 우승하고 싶어 한다.

연못의 폭과 각 칸에 있는 물고기의 가치, 그물의 가로의 길이와 세로의 길이가 주어질 때, 잡을 수 있는 물고기의 최대이득을 구하는 프로그램을 작성하시오.

입력

첫 번째 줄에 연못의 폭 N이 입력된다. (N <= 100 인 자연수)
두 번째 줄에 그물의 폭 W가 입력된다. (W <= N 인 자연수)
세 번째 줄 W개의 물고기의 가치가 공백으로 구분되어 주어진다. 각 물고기의 가치는 7이하의 자연수이다. 0일 경우에는 물고기가 없다는 의미이다.

출력

잡을 수 있는 물고기의 최대 가치를 출력한다.

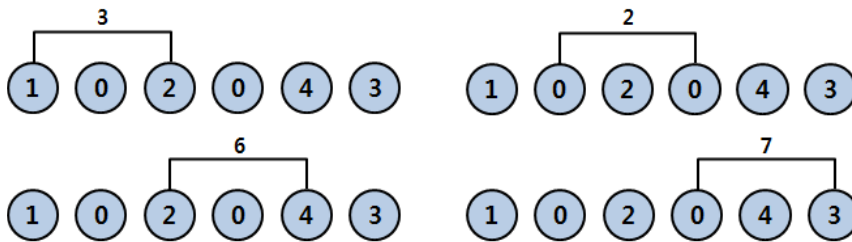
입력 예	출력 예
6 3 1 0 2 0 4 3	7

풀이

이 문제는 전체탐색법을 이용하여 간단하게 해결할 수 있다. 폭이 N인 연못에서 폭이 W인 그물을 던졌을 때 최대 이득을 얻을 수 있는 구간을 찾는 문제이다. 가장 단순한 방법으로 N개의 주어진 수들 중 연속된 W개의 수들을 탐색하여 합을 구한 다음 최댓값을 갱신하는 방법으로 접근할 수 있다.

먼저 첫 번째 데이터부터 탐색하여 W개의 합을 구한 다음 최댓값을 갱신하고, 두 번째 데이터부터 탐색하여 W개의 합을 구하여 최댓값을 갱신한다. 이런 방법으로 모든 구간을 전체탐색법으로 확인할 수 있다.

입출력 예의 경우 다음과 같은 과정으로 해를 구해나간다.



따라서 위의 경우 해는 7이 된다.

위의 과정을 보면 탐색을 시작하는 지점이 0번으로부터 시작하여 1씩 증가하는 것을 알 수 있으며, 시작점을 지정하면 그물의 폭인 W만큼 탐색을 진행한다. 따라서 마지막 탐색의 시작 지점은 $N-W+1$ 이 된다. 핵심 탐색 부분을 구현하면 다음과 같다.

줄	코드	참고
1	`for(int i=0; i<N-W+1; i++)`	
2	`{`	
3	`  for(int j=0; j<W; j++)`	
4	`    printf("%d ", i+j);`	
5	`  puts("");`	
6	`}`	

$n=8$ 이고 $w=5$ 일 때, 위 탐색방법의 출력결과는 결과는 다음과 같다.

```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
```

즉, [0, 4] 구간, [1, 5]구간, [2, 6]구간, [3, 7]구간으로 모두 4번을 검사한다.

위 소스코드에서 $N-W+1$ 을 생각하기 어려운 경우에는 배열을 좀 더 크게 잡은 후 다음과 같이 작성해도 관계없다.

```
for(int i=0; i<N; i++)
{
  for(int j=0; j<W; j++)
    printf("%d ", i+j);
  puts("");
}
```

위와 같이 작성하면 생각하기 쉽기 때문에 빠른 시간에 코딩이 가능하다. 위의 코드의 출력결과는 다음과 같다.

0 1 2 3 4

1 2 3 4 5

2 3 4 5 6

3 4 5 6 7

4 5 6 7 **8**

5 6 7 **8 9**

6 7 **8 9 10**

7 **8 9 10 11**

빨간색으로 표시된 부분은 실제로 데이터가 0이 들어있어 구간의 합을 구하더라도 해를 구하는데 영향을 미치지 않는다.

위의 아이디어 들을 이용하여 문제를 해결한 소스코드는 다음과 같다.

```
#include <stdio.h>

int data[101], N, W, ans = 0;

int main()
{
  scanf("%d%d", &N, &W);
  for(int i=0; i<N; i++)
    scanf("%d", data+i);

  for(int i=0; i<N+W-1; i++)
  {
    int sum = 0;
    for(int j=0; j<W; j++)
      sum+=data[i+j];
    if(sum>ans) ans=sum;
  }

  printf("%d", ans);
}
```

이 알고리즘의 계산량은 1~N의 각 위치에 대해서 W만큼 탐색을 하므로 $O(NW)$가 됨을 알 수 있다.

문제에서 제시한 N의 최대치가 100,000이 입력되고 그물의 크기가 적당히 크면 수행시간이 많이 걸리므로 좀 더 효율적인 알고리즘이 필요하다.

문제 5

고기잡이(L)

우리나라 최고의 어부 정올이가 이번에 네모네모 배 고기잡이 대회에 참가한다.

이 대회에는 3개의 라운드가 있는데, 두 번째 라운드는 2차원 형태로 표현될 수 있는 작은 연못에서 길쭉한 그물을 던져서 최대한 많은 고기를 잡는 것이 목적이다.

1라운드의 예를 들면 연못의 크기가 1*6이고 물고기의 위치와 가치가 다음과 같다고 하자.

1 0 2 0 4 3

여기서 그물의 크기는 1*3이라고 할 때, 잡을 수 있는 방법은 (1 0 2), (0 2 0), (2 0 4), (0 4 3)의 4가지 방법이 있다.

이 중 가장 이득을 보는 방법은 마지막 방법 0 + 4 + 3 = 7이다. 따라서 주어진 경우의 최대 이득은 7이 된다. 정올이는 최대한 가치가 큰 물고기를 잡아서 우승하고 싶어 한다.

연못의 폭과 각 칸에 있는 물고기의 가치, 그물의 가로의 길이가 주어질 때, 잡을 수 있는 물고기의 최대이득을 구하는 프로그램을 작성하시오.

입력

첫 번째 줄에 연못의 폭 N, M이 입력된다. (N, M <= 100 인 자연수)
두 번째 줄에 그물의 폭 W, H가 입력된다. (W <= N, H <= M 인 자연수)
세 번째 줄에 N*M개의 물고기의 가치가 공백으로 구분되어 주어진다. 각 물고기의 가치는 7 이하의 자연수이다. 0일 경우에는 물고기가 없다는 의미이다.

출력

잡을 수 있는 물고기의 최대 가치를 출력한다.

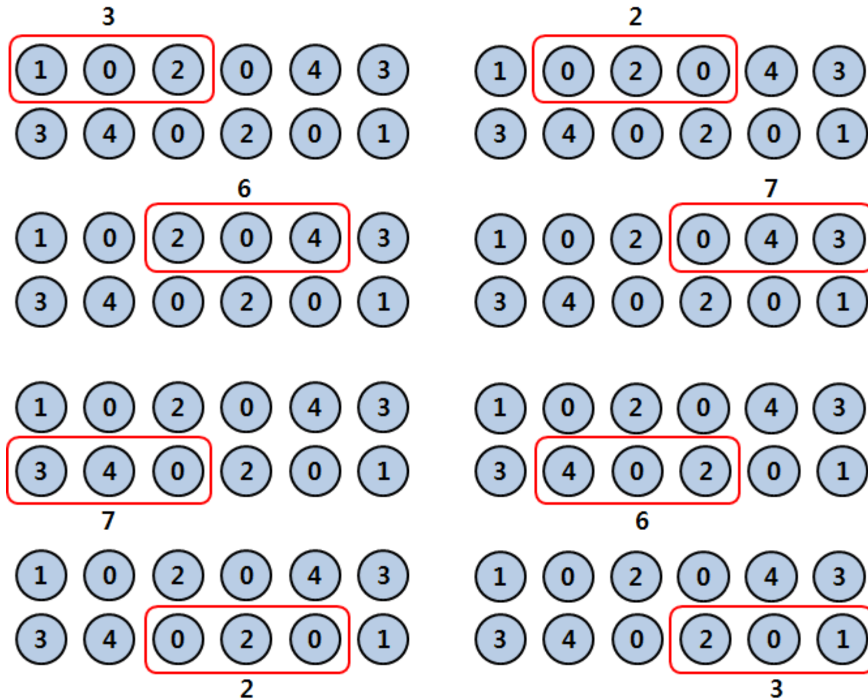
입력 예	출력 예
2 6 1 3 1 0 2 0 4 3 3 4 0 2 0 1	7

풀이

[문제 4] 고기잡이(S)가 2차원으로 확장된 형태의 문제이다. 따라서 이 문제 *N*, *M*값이 크지 않으므로 2차원 전체탐색법을 이용하여 해결할 수 있다.

먼저 첫 번째 줄의 데이터부터 탐색하여 *W*H* 개의 합을 구한 다음 최댓값을 갱신하고, 그 다음 순서대로 탐색하여 *W*H* 개의 합을 구하여 최댓값을 갱신한다. 이런 방법으로 (1,1)~(N, M) 구간까지 확인한다.

입출력 예시(N=2, M=6, W=1, H=3)를 예로 들면, 다음과 같다.



이 과정을 거쳐 최댓값이 7임을 알 수 있다. 구하는 과정은 앞에서 다룬 것과 대부분 동일하다. 소스코드는 다음과 같다.

```c
#include <stdio.h>

int data[100][100];
int N, M, H, W;
int res=0;

int main()
{
  scanf("%d %d", &N, &M);
  scanf("%d %d", &H, &W);
    for(int i=0; i<N; i++)
      for(int j=0; j<M; j++)
         scanf("%d", &data[i][j]);

  for(int i=0; i<N-H+1; i++)
  {
    for(int j=0; j<M-W+1; j++)
    {
      int sum = 0;
      for(int a=0; a<H; a++)
        for(int b=0; b<W; b++)
          sum+=data[i+a][j+b];
      if(sum>res) res=sum;
    }
  }
  printf("%d", res);
}
```

위 두 알고리즘의 계산량은 N*M의 위치에 대해서 H*W만큼 탐색을 하므로 $O(NMHW)$가 됨을 알 수 있다.

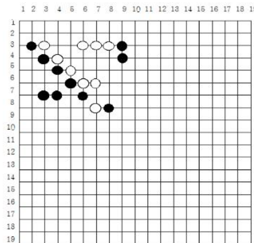
N*M의 최대치가 300,000이고 그물의 크기가 적당히 크면 수행시간이 매우 많이 걸린다.

1차원에서는 전체탐색을 해도 2중 반복문이 되지만 2차원에서는 4중 반복문이 되므로 매우 비효율적인 알고리즘이다. N, M의 값이 조금만 커져도 제한 시간 이내에 해를 구할 수 없다.

문제 6

오목

오목은 바둑판에 검은 바둑알과 흰 바둑알을 교대로 놓아서 겨루는 게임이다. 바둑판에는 가로, 세로 19개의 선으로 이루어져 있다.



오목은 위의 그림에서와 같이 같은 색의 바둑알이 연속적으로 다섯 알이 놓이면 그 색이 이기게 된다. 여기서 연속적이란 가로, 세로 또는 대각선 방향 모두를 뜻한다.

즉, 위의 그림은 검은색이 이긴 경우이다. 하지만 여섯 알 이상이 연속적으로 놓인 경우에는 이긴 것이 아니다. 입력으로 바둑판의 어떤 상태가 주어졌을 때, 검은색이 이겼는지, 흰색이 이겼는지 또는 아직 승부가 결정되지 않았는지를 판단하는 프로그램을 작성하시오.

단, 검은색과 흰색이 동시에 이기거나 검은색 또는 흰색이 두 군데 이상에서 동시에 이기는 경우는 입력으로 들어오지 않는다.

입력

입력 파일은 19줄에 각 줄마다 19개의 숫자로 표현되는데, 검은 바둑알은 1, 흰 바둑알은 2, 알이 놓이지 않은 자리는 0으로 표시되며, 숫자는 한 칸씩 띄어서 표시된다.

출력

첫 번째 줄에 검은색이 이겼을 경우에는 1을, 흰색이 이겼을 경우에는 2를, 아직 승부가 결정되지 않았을 경우에는 0을 출력한다. 검은색 또는 흰색이 이겼을 경우에는 둘째 줄에 연속된 다섯 개의 바둑알 중에서 가장 왼쪽에 있는 바둑알(연속된 다섯 개의 바둑알이 세로로 놓인 경우, 그중 가장 위에 있는 것)의 가로줄 번호와 세로줄 번호를 순서대로 출력한다.

입력 예	출력 예
위 그림과 같은 경우	1 3 2

출처: 한국정보올림피아드(2003 전국본선 초등부)

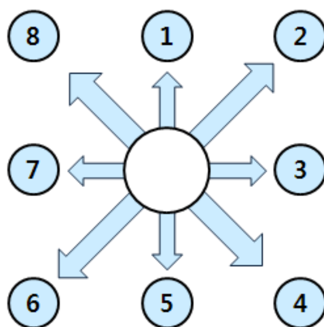
풀이

다양한 경우에 대한 처리를 연습하기에 좋은 문제이다. 우리가 흔히 접할 수 있는 오목 게임에서 흑과 백이 순서를 번갈아 가며 돌을 놓게 되는데 매 순간마다 승패를 검사해야 한다. 이 때 사용되는 알고리즘이 바로 이 문제가 요구하는 것이다. 게임의 규칙은 이미 잘 알고 있으므로 생략하고 어떻게 승패를 검사할 것인지에 대해 고민해 보자.

일단 기본적으로 바둑판을 2차원 배열로 생각하고 2차원 구조로 전체 탐색을 진행하는 것은 당연하다. 전체 탐색을 진행하면서 현재 탐색 중인 돌을 기준으로 오목검사를 행하는 것이 가장 일반적인 아이디어다. 하지만 이 검사에 대해서 생각해야 될 사항들이 많다. 기본적인 탐색 아이디어는 다음과 같다.

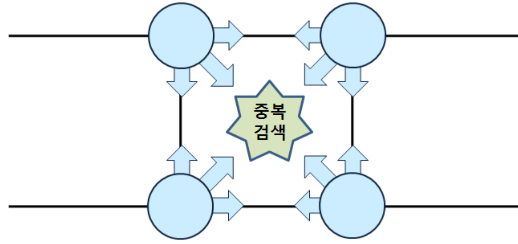
1. (0, 0)부터 (18, 18)까지 행 우선으로 탐색을 시작한다.
2. 현재 탐색 중인 돌을 기준으로 오목이 완성되었는지 검사한다.
3. 완성되지 않았으면 탐색을 진행하기 위해 2번으로 간다.
4. 완성된 돌의 위치와 색깔을 출력한다.

이 문제 해결의 핵심은 2번 항목의 오목이 완성되었는지 검사하는 부분이다. 이 부분은 기본적으로 탐색 중인 돌을 기준으로 아래 그림과 같이 8방향에 대해서 생각해볼 수 있다.



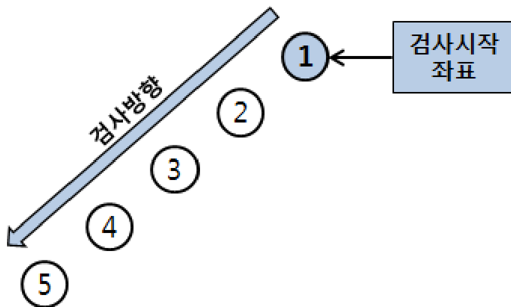
그리고 해당 방향에 돌이 있다면 돌의 개수를 세어야 될 것이다. 하지만 생각한 것보다 다양한 경우가 발생되며, 처리해야할 것들이 많다.

먼저 위 그림과 같은 8방향 검사 알고리즘을 적용하면 다음과 같이 중복으로 검사되는 경우가 발생한다.

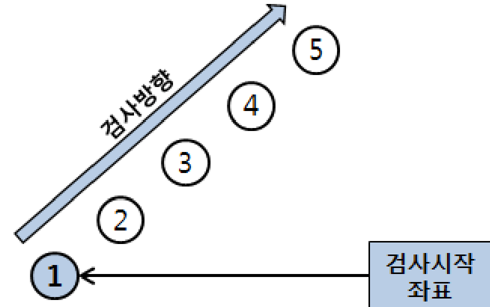


따라서 (0, 0)에서부터 순차적으로 탐색해 나간다면, 8방향 검사를 다 할 필요가 없다. 즉, 검사를 마친 좌표에 대해서는 굳이 검사를 할 필요가 없는 것이다. 그렇다면 8방향 중 몇 곳을 조사해야 할까?

문제에서 요구하는 좌표가 연속된 다섯 개의 바둑알 중에서 가장 왼쪽에 있는 바둑알의 좌표를 요구하므로 검사되는 돌의 좌표에서 오른쪽 방향으로 검사가 이뤄져야 한다.



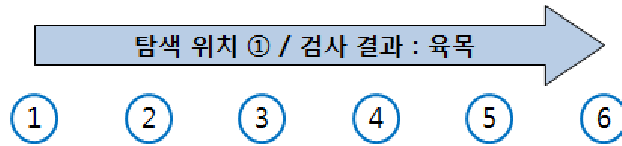
① ↙방향 검사는 검사 시작 돌이 가장 왼쪽 좌표가 될 수 없다.



②↗방향 검사는 검사 시작 돌이 가장 왼쪽 좌표가 될 수 있다.

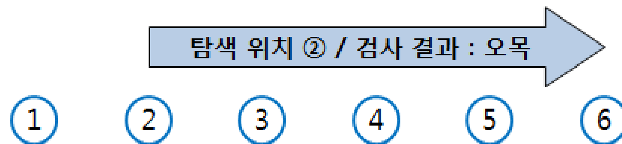
위 그림에서 보는 것과 같이 ↙방향보다는 ↗방향을 선택해야 검사되는 돌이 가장 왼쪽 좌표가 된다. 나머지 방향에 대해서는 별로 애매한 부분이 없을 것이다. 즉, ↓, ↘, →, ↗ 4방향을 선택하는 것이 현명하다.

이와 같이 4방향을 선택하면 된다. 하지만 또 다른 문제점이 발생할 수 있다. 다음 경우를 확인해 보자.



① 연속되어 같은 돌이 6개 놓인 경우 (육목)이므로 검사결과 오목이 아님

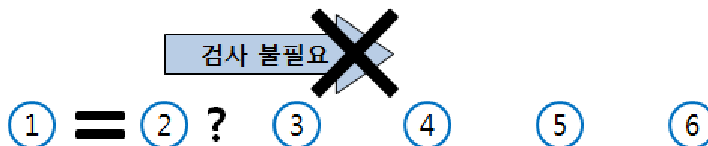
여기까진 괜찮으나 다음 좌표 위치에서는 어떤 결과가 발생할까?



① 2번 돌에서 검사결과 오목임

1번 돌이 있어서 육목인데 ← 방향은 검사하지 않으므로 위와 같이 잘못된 결과가 발생할 수 있다. 따라서 올바른 검사 결과가 나오게 하기 위해서는 ←방향에 같은 돌이 있으면 → 방향은 검사를 하지 말아야 한다.

일반화 시키면 검사 방향의 정반대 방향에 같은 돌이 있으면 검사방향쪽으로 검사를 하지 말아야 하는 것이다. 이전의 검사에서 그 방향은 이미 검사했기 때문이다.



이와 같이 이 문제에서는 다양한 경우에 대해 연습할 수 있는 좋은 문제이다. 하지만 이런 문제는 스스로 소스코드가 맞는지 판단하기가 어려우니 반드시 online judge 등에서 판정을 받아보는 것이 좋다.

전체 소스는 다음과 같다.

줄 | 코드 | 참고

```
1  #include<stdio.h>
2  int a[19+2][19+2];
3  
4  int main()
5  {
6    int i, j;
7    for(i=1; i<=19; i++)
8      for(j=1; j<=19; j++)
9        scanf("%d", &a[i][j]);
10   for(i=1; i<=19; i++)
11     for(j=1; j<=19; j++)
12       if(a[i][j]!=0)
13       {
14         if(a[i][j-1]!=a[i][j] && search1(a[i][j], i, j, 1)==1)
15         {
16           printf("%d\n%d %d", a[i][j], i, j);
17           return 0;
18         }
19         if(a[i-1][j-1]!=a[i][j] && search2(a[i][j], i, j, 1)==1)
20         {
21           printf("%d\n%d %d", a[i][j], i, j);
22           return 0;
23         }
24         if(a[i-1][j]!=a[i][j] && search3(a[i][j], i, j, 1)==1)
25         {
26           printf("%d\n%d %d", a[i][j], i, j);
27           return 0;
28         }
29         if(a[i+1][j-1]!=a[i][j] && search4(a[i][j], i, j, 1)==1)
30         {
31           printf("%d\n%d %d", a[i][j], i, j);
32           return 0;
33         }
34       }
35   printf("0");
36   return 0;
37 }
```

참고:
2: 19×19, 사방의 끝을 0으로 채움
12: 돌이 놓여있는 경우에
35: 무승부인 경우

줄	코드	참고
1	`int search1(int color, int i, int j, int cnt)`	1: → 방향 7: ↘ 방향 14: ↓ 방향 21: ↗ 방향
2	`{`	
3	`  for(; color==a[i][j+1]; j++)`	
4	`    cnt++;`	
5	`  return cnt==5 ? 1:0;`	
6	`}`	
7	`int search2(int color, int i, int j, int cnt)`	
8	`{`	
9	`  for(; color==a[i+1][j+1]; i++, j++)`	
10	`    cnt++;`	
11	`  return cnt==5 ? 1:0;`	
12	`}`	
13		
14	`int search3(int color, int i, int j, int cnt)`	
15	`{`	
16	`  for(; color==a[i+1][j]; i++)`	
17	`    cnt++;`	
18	`  return cnt==5 ? 1:0;`	
19	`}`	
20		
21	`int search4(int color, int i, int j, int cnt)`	
22	`{`	
23	`  for(; color==a[i-1][j+1]; i--,j++)`	
24	`    cnt++;`	
25	`  return cnt==5 ? 1:0;`	
26	`}`	

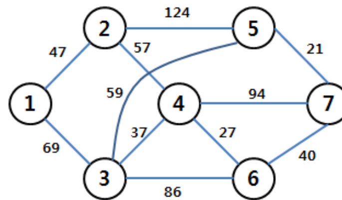
문제 7

연구활동 가는 길(S)

정올이는 GSHS에서 연구활동 교수님을 뵈러 A대학교를 가려고 한다. 출발점과 도착점을 포함하여 경유하는 지역 n개, 한 지역에서 다른 지역으로 가는 방법이 총 m개이며 GSHS는 지역 1이고 A대학교는 지역 n이라고 할 때 대학까지 최소 비용을 구하시오.

단, n은 10 이하, m은 30 이하, 그리고 한 지역에서 다른 지역으로 가는 데에 필요한 비용은 모두 200 이하 양의 정수이며 한 지역에서 다른 지역으로 가는 어떠한 방법이 존재하면 같은 방법과 비용을 통해 역방향으로 갈 수 있다.

다음 그래프는 예를 보여준다.(단, 정점a->정점b로의 간선이 여러 개 있을 수 있으며, 자기 자신으로 가는 정점을 가질 수도 있다.)



최소 비용이 드는 경로 : 1→3→5→7, 최소 비용 : 69+59+21=149

입력

첫 번째 줄에는 정점의 수 n과 간선의 수 m이 공백으로 구분되어 입력된다. 다음 줄부터 m개의 줄에 걸쳐서 두 정점의 번호와 가중치가 입력된다. (자기 간선, 멀티 간선이 있을 수 있다.)

출력

대학까지 가는 데 드는 최소 비용을 출력한다. 만약 갈 수 없다면 "-1"을 출력.

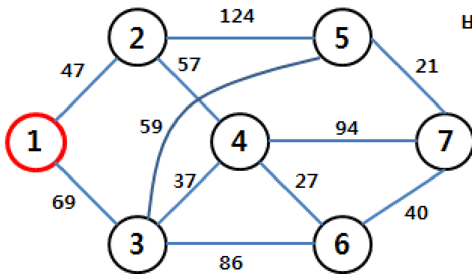
입력 예	출력 예
7 11 1 2 47 1 3 69 2 4 57 2 5 124 3 4 37 3 5 59 3 6 86 4 6 27 4 7 94 5 7 21 6 7 40	149

풀이

이 문제는 그래프 상의 최단경로를 구하는 매우 유명한 문제이다. 이 문제를 해결하는 알고리즘은 여러 가지가 알려져 있지만, 어려운 알고리즘을 모르더라도 전체탐색법을 통하여 해결할 수 있다.

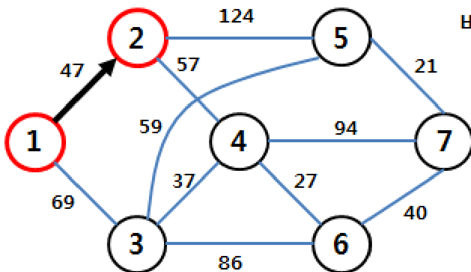
이 문제는 그래프 구조이므로 비선형탐색법으로 해를 구할 수 있다. 먼저 출발정점에서 깊이우선탐색을 이용하여 출발점으로부터 도착점까지 가능한 모든 경로에 대해서 구해본다. 하나의 경로를 구할 때마다 해를 갱신하면서 최종적으로 가장 적합한 해를 출력한다.

주어진 예를 통하여 전체탐색하는 과정을 간단하게 살펴보자.



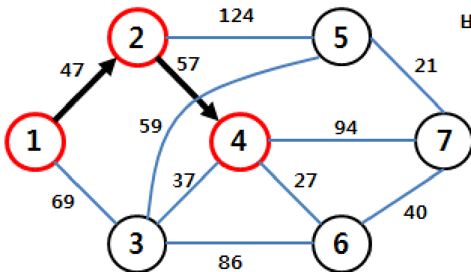
처음 출발점에서 2, 3의 정점 중 2번 정점으로 출발한다.

현재까지 구한 최소 이동거리 = ∞



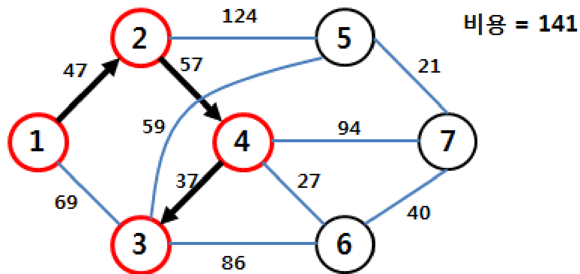
현재 2번 정점까지 이동거리 47, 다음으로 갈 수 있는 4, 5 중 4번을 먼저 탐색.

현재까지 구한 최소 이동거리 = ∞



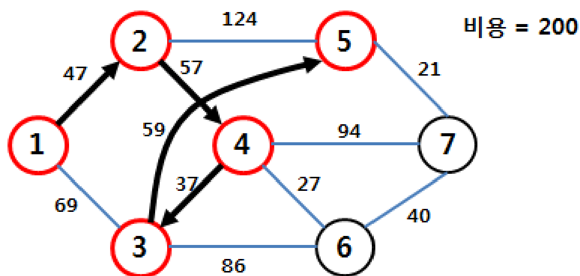
4번 정점까지 104를 이동한 후, 3, 6, 7중 먼저 3으로 먼저 탐색.

현재까지 구한 최소 이동거리 = ∞



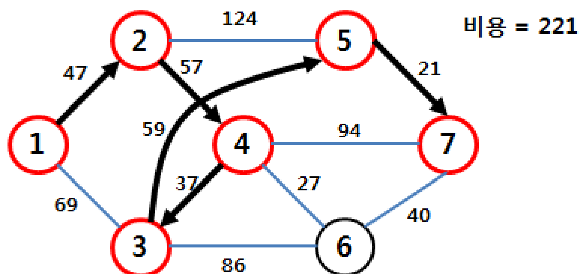
3으로 이동한 후, 후 다시 5번으로 이동.

현재까지 구한 최소 이동거리 = ∞



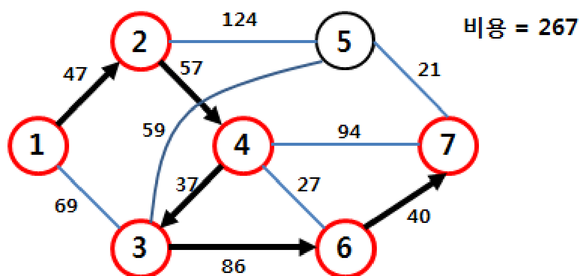
5번까지 이동 후, 7번으로 이동

현재까지 구한 최소 이동거리 = ∞



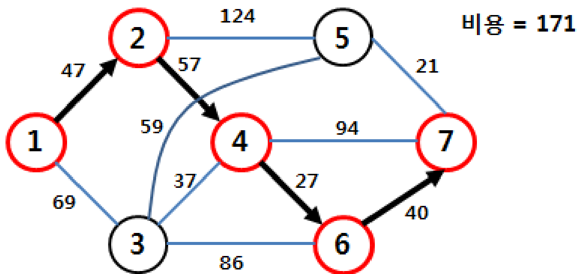
첫 번째 경로 찾음. 총 비용 221이므로 현재까지 구한 최소 이동거리는 221로 갱신

현재까지 구한 최소 이동거리 = 221



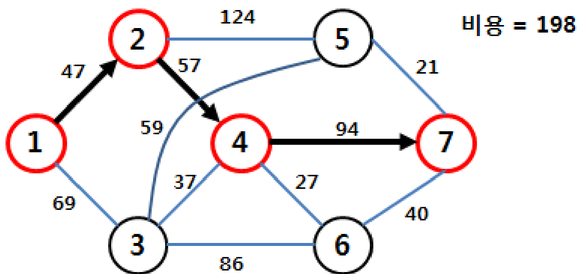
두 번째로 구한 경로는 267이 된다. 이 해는 지금까지의 해보다 더 크므로 갱신하지 않는다.

현재까지 구한 최소 이동거리 = 221



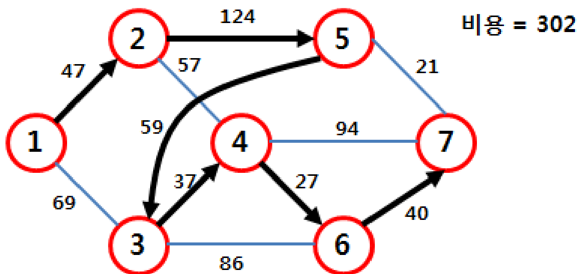
세 번째로 구한 해는 171이 된다.

현재까지 구한 최소 이동거리 = 171



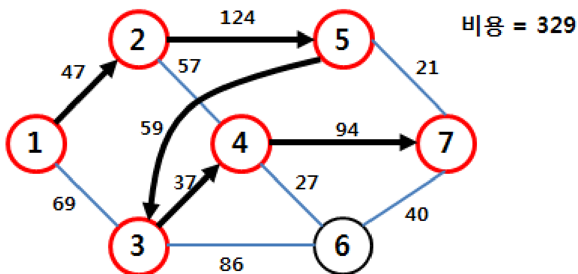
다음으로 구한 해는 198이 된다. 이 해는 지금까지 해보다 더 크므로 갱신하지 않는다.

현재까지 구한 최소 이동거리 = 171



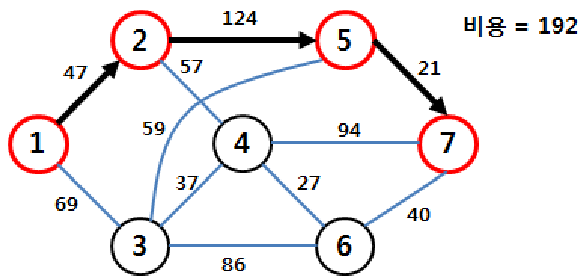
다음으로 구한 해는 302가 된다. 이 해는 지금까지 해보다 더 크므로 갱신하지 않는다.

현재까지 구한 최소 이동거리 = 171



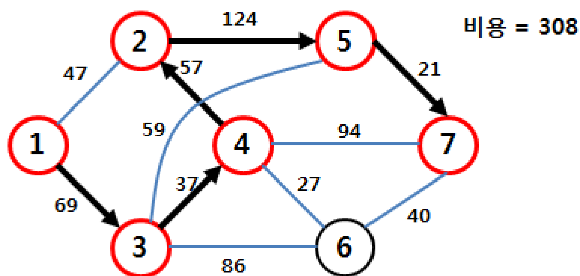
다음으로 구한 해는 329가 된다. 이 해는 지금까지 해보다 더 크므로 갱신하지 않는다.

현재까지 구한 최소 이동거리 = 171



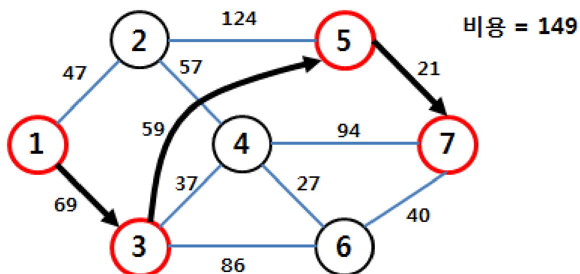
다음으로 구한 해는 192가 된다. 이 해는 지금까지 해보다 더 크므로 갱신하지 않는다.

현재까지 구한 최소 이동거리 = 171



다음으로 구한 해는 308이 된다. 이 해는 지금까지 해보다 더 크므로 갱신하지 않는다.

현재까지 구한 최소 이동거리 = 171



다음으로 구한 해는 149가 된다. 이 해는 지금까지 해보다 작으므로 갱신한다.

현재까지 구한 최소 이동거리 = 149

따라서 위의 경우 전체탐색법으로 탐색한 결과 최소 이동거리는 149가 됨을 알 수 있다. 위의 과정과 같은 방법으로 코딩한 결과는 다음과 같다.

줄	코드	참고
1	`#include<stdio.h>`	
2	`int n, m, G[11][11], sol = 0x7fffffff, chk[11];`	
3		
4	`void solve(int V, int W)`	
5	`{`	
6	`  if(V==n)`	

줄	코드	참고
7	`  {`	
8	`    if(W<sol) sol=W;`	
9	`      return;`	
10	`  }`	
11	`  for(int i=1; i<=n; i++)`	
12	`    if(!chk[i] && G[V][i])`	
13	`    {`	
14	`      chk[i]=1;`	
15	`      solve(i, W+G[V][i]);`	
16	`      chk[i]=0;`	
17	`    }`	
18	`}`	
19	`int main(void)`	
20	`{`	
21	`  scanf("%d %d", &n, &m);`	
22	`  for(int i=0; i<m; i++)`	
23	`  {`	
24	`    int s, e, w;`	
25	`    scanf("%d %d %d", &s, &e, &w);`	
26	`    G[s][e]=G[e][s]=w;`	
27	`  }`	
28	`  solve(1, 0);`	
29	`  printf("%d\n", sol==0x7fffffff ? -1:sol);`	
30	`  return 0;`	
31	`}`	

이 문제의 경우 정점과 간선의 수가 많지 않으므로 인접행렬로도 충분히 처리가 가능하기 때문에 인접행렬로 처리한다.

solve(a, b)는 현재 a정점까지 방문한 상태로 이동거리가 b라고 정의하고 있으며, chk배열이 현재까지 방문한 정점들의 정보를 가지고 있다. 다음 정점으로 진행할 때 14행과 같이 chk배열에 다음 방문할 정점을 체크하고 만약 백트랙해서 돌아온다면, 16행과 같이 체크를 해제하며 전체탐색을 진행한다.

6행에서 도착 여부를 확인하여 현재 정점이 도착점이라면, 지금까지의 이동 거리와 현재까지 구한 해를 비교하여 더 좋은 해가 있으면 해를 갱신한다. 이와 같이 작성할 경우 도시의 수가 n개라고 할 때 $O(n!)$의 계산이 필요하다.

문제 8

리모컨

컴퓨터실에서 수업 중인 정보 선생님은 냉난방기의 온도를 조절하려고 한다.

냉난방기가 멀리 있어서 리모컨으로 조작하려고 하는데, 리모컨의 온도 조절 버튼은 다음과 같다.

1) 온도를 1도 올리는 버튼
2) 온도를 1도 내리는 버튼
3) 온도를 5도 올리는 버튼
4) 온도를 5도 내리는 버튼
5) 온도를 10도 올리는 버튼
6) 온도를 10도 내리는 버튼

이와 같이 총 6개의 버튼으로 목표 온도를 조절해야 한다.

현재 설정 온도와 변경하고자 하는 목표 온도가 주어지면 이 버튼들을 이용하여 목표 온도로 변경하고자 한다.

이 때 버튼 누름의 최소 횟수를 구하시오. 예를 들어, 7도에서 34도로 변경하는 경우,

7 → 17 → 27 → 32 → 33 → 34

이렇게 총 5번 누르면 된다.

입력

현재 온도 a와 목표 온도 b가 입력된다(0 <= a, b <= 40).

출력

최소한의 버튼 사용으로 목표 온도가 되는 버튼 누름의 횟수를 출력한다.

입력 예	출력 예
7 34	5

풀이

이 문제는 시작 온도에서 목표 온도로 되는 과정에서 버튼의 최소 이용 횟수를 구하는 문제이다. 먼저 전체탐색법으로 문제를 해결해 보자.

```
#include <stdio.h>

int a, b;
int res=40;

void f(int temp, int cnt)
{
  if(cnt>res) return ;
  if(temp==b)
  {
    if(cnt<res) res=cnt;
    return;
  }
  f(temp+10,cnt+1);f(temp+5,cnt+1);f(temp+1,cnt+1);
  f(temp-10,cnt+1);f(temp-5,cnt+1);f(temp-1,cnt+1);
}

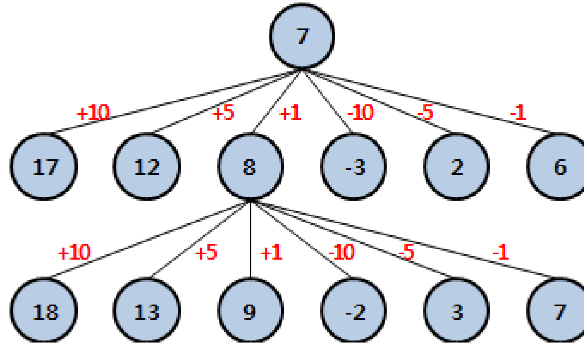
int main()
{
  scanf("%d%d", &a, &b);
  f(a, 0);
  printf("%d", res);
  return 0;
}
```

res의 초깃값을 40으로 설정한 이유는 입력의 정의역이 0~40이기 때문이며 최악의 경우 40번을 눌러야하기 때문에 최댓값으로 설정해 놓은 것이다. 이것은 다음에 백트래킹 함수 f의 호출 한계와도 관련이 있다.

백트래킹 함수의 의미는 다음과 같다.

f(temp, cnt) = 온도가 temp일 때, 버튼 누름 횟수는 cnt

각 온도에 대해서 다음으로 누르는 버튼은 +10도, +5도, +1도, −10도, −5도, −1도로 전체 탐색한다.



이렇게 탐색하여 목표 온도에 도달하면 탐색을 종료한다. 이 탐색 버튼의 수 6과 목표 온도에 도달하는 최악의 횟수 res에 비례하므로 계산량은 $O(6^{res})$ 이다. res를 40으로 설정하였기 때문에 계산량이 많아 제한 시간 안에 해결하기 어렵다. 따라서 계산량을 줄이기 위한 탐색영역을 배제할 필요가 있다.

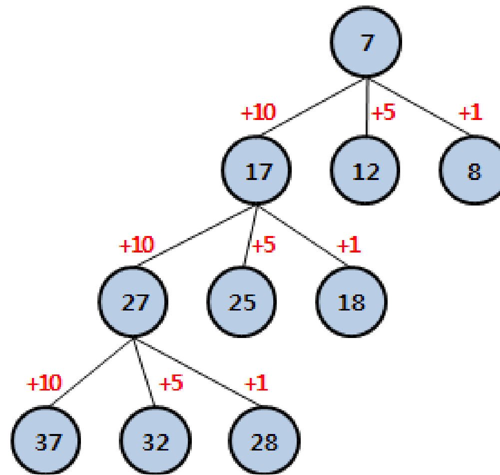
먼저 간단한 수학적인 상식을 적용시켜보면,

현재 온도가 목표 온도보다 작은 경우, 온도를 내릴 필요가 없다.
반대로, 현재 온도가 목표 온도보다 큰 경우, 온도를 올릴 필요가 없다.

이 원리를 이용하면 위 소스코드의 14~15행의 탐색범위를 줄여서 다음과 같이 표현할 수 있다.

줄	코드	참고
14	`if(temp<b){`	
15	`  f(temp+10,cnt+1);f(temp+5,cnt+1);f(temp+1,cnt+1);`	
16	`}`	
17	`else{`	
18	`  f(temp-10,cnt+1);f(temp-5,cnt+1);f(temp-1,cnt+1);`	
19	`}`	

위 소스를 토대로 실행하면 다음 호출 구조로 줄일 수 있다.



앞의 소스에 비해 계산량을 정확히 반으로 줄인 $O(3^{res})$ 결과를 볼 수 있다. 하지만 여전히 res의 최악의 횟수가 계산량에 영향을 많이 미치기 때문에, res를 줄이는 수학적 검증이 필요하다.

또 최악의 횟수를 줄일 수 있지만 현재 온도와 목표 온도의 차이가 아주 많이 나는 경우에는 여전히 계산량이 많아진다. 따라서 res를 조절하는 것 보다 계산 과정 중 중복된 연산을 줄이고, 목표 온도에 도달한 경우 더 이상 함수를 호출하지 않는 방법을 선택하는 것이 좋다.

백트래킹은 모든 호출이 끝이 나야만 해답을 찾을 수 있기 때문에, 이러한 깊이우선 탐색은 이 문제에서 비효율적이다. 이 방법 대신 너비우선탐색 기법을 이용하여 버튼 누름 횟수를 기준으로 가장 먼저 목표 온도에 도달할 때까지 탐색하고 중단하는 방법으로 설계해 보자.

코드	참고
`#include <stdio.h>`	
`#include <queue>`	
`using namespace std;`	
`struct ELE{int v, cnt;};`	
`queue<ELE> Q;`	

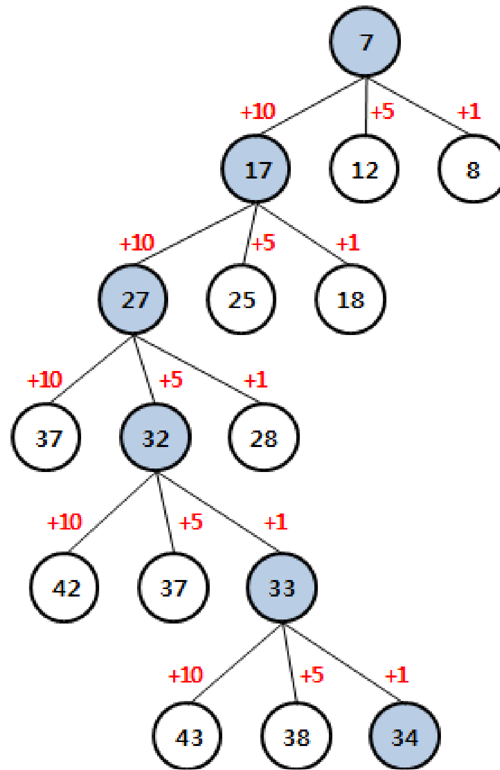
줄	코드	참고

```
8   int main()
9   {
10    int a, b, i;
11    ELE temp;
12    scanf("%d %d",&a,&b);
13    Q.push({a, 0});
14    while(!Q.empty())
15    {
16      temp = Q.front(), Q.pop();
17      if(temp.v == b)
18        break;
19      if(temp.v<b)
20      {
21        Q.push({temp.v+10, temp.cnt+1});
22        Q.push({temp.v+5, temp.cnt+1});
23        Q.push({temp.v+1, temp.cnt+1});
24      }
25      else
26      {
27        Q.push({temp.v-10, temp.cnt+1});
28        Q.push({temp.v-5, temp.cnt+1});
29        Q.push({temp.v-1, temp.cnt+1});
30      }
31    }
32    printf("%d", temp.cnt);
33  }
```

앞의 백트래킹 소스를 너비우선탐색으로 변형한 코드이다. 너비우선탐색 알고리즘은 목표 온도 b에 도달하면 곧 바로 다른 모든 탐색을 중지하고 결과를 얻을 수 있어, 불필요한 탐색을 막을 수 있고 계산량을 확실히 줄일 수 있다.

소스에 대해 잠깐 설명하면 2행에서 STL queue 라이브러리를 이용하고 있다. 5행에서 ELE라는 구조체 형식의 Q를 선언하고, 13~31행에서 너비우선탐색 알고리즘을 사용하고 있다. 13행, 21~23행, 27~29행에서 구조체를 큐에 넣을 때 중괄호{, }를 이용하여 한 번에 대입할 수 있음을 참고하기 바란다.

7에서 34도로 변하는 과정을 트리로 표현한 결과이다.



34도에 도달하게 되면 모든 연산을 중지하고 결과를 얻을 수 있다. 너비우선탐색으로 탐색하여 목표 온도에 도달하는 경우 계산량은 이 탐색 버튼의 수 3과 목표 온도에 도달하는 최적의 횟수 cnt에 비례하므로 계산량을 줄일 수 있다.

문제 9

오른편 절단 가능 소수

수학자들에게 소수란 매우 흥미 있는 연구 주제이다. 소수(prime number)란 약수가 1과 자기 자신밖에 없는 1보다 큰 자연수를 말한다. 수학자들은 소수를 연구하면서 특이한 소수들을 발견하여 이름을 명명하였다. 메르센 소수, 페르마 소수, 쌍둥이 소수 등이 그 예이다.

우리에게는 생소하지만 오른편 절단 가능 소수가 있다. 이 소수는 오른쪽부터 하나씩 제거해도 계속 소수가 되는 소수이다.

크기가 네 자리인 7193을 예로 들어보자. 7193은 소수이고, 7193의 오른편 숫자 3을 제거하여 남은 719도 소수이다. 719의 오른편 숫자 9를 제거하여 남은 71도 소수이다. 71의 오른편 숫자 1을 제거하여 남은 7도 소수이다. 따라서 7193은 오른편 절단 가능 소수이다.

입력

자릿수 n이 정수로 입력된다.(1 <= n <= 10)

출력

1. n자리로 이루어진 오른편 절단 가능 소수들을 한 줄에 하나씩 오름차순으로 출력한다.
2. 마지막 줄에 출력된 오른편 절단 가능 소수들의 개수를 출력한다.

입력 예	출력 예
2	23 29 31 37 53 59 71 73 79 9

풀이

이 문제는 n자리의 숫자들 중 오른편 절단 가능 소수를 찾고 그 개수를 출력하는 문제이다. 먼저 길이가 n인 순열을 생성하고, 소수인지 판별해보는 전체탐색법으로 문제를 해결해 보자.

코드	참고

```
#include<stdio.h>

int n, cnt;
int isprime(int x)
{
  if(x<2) return 0;
  for(int i=2; i*i<=x; i++)
    if(x%i==0)
      return 0;
  return 1;
}

void f(int num, int len)
{
  if(len==n)
  {
    if(num==0) return ;

    int flag=1;
    int temp=num;
    while(temp)
    {
      if(!isprime(temp))
        return ;
      temp /= 10;
    }
    cnt++;
    printf("%d\n", num);
    return ;
  }
  else
  {
    for(int i=1; i<=9; i++)
```

줄	코드	참고
34	`        f(num*10+i, len+1);`	
35	`    }`	
36	`}`	
37		
38	`int main()`	
39	`{`	
40	`   scanf("%d", &n);`	
41	`   f(0, 0);`	
42	`  printf("%d", cnt);`	
43	`}`	

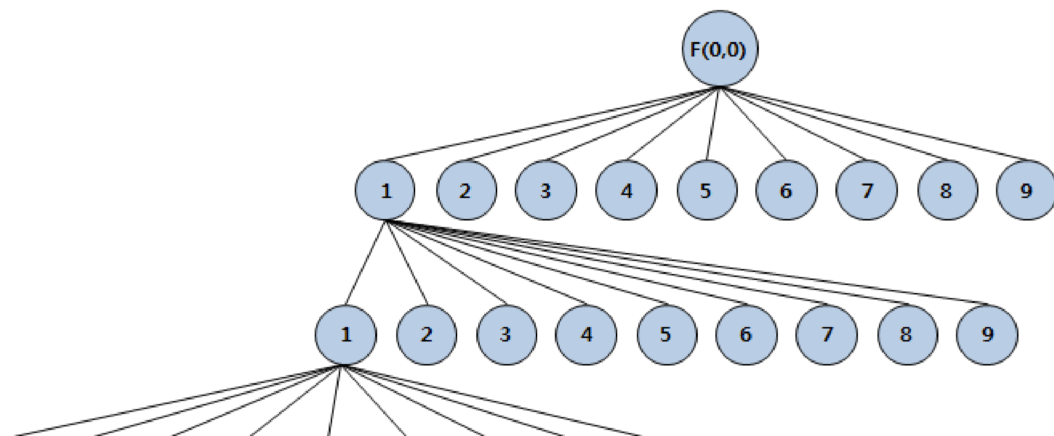
4~11행의 isprime(x)함수는 정수 x가 소수이면 1을 리턴, 소수가 아니면 0을 리턴하는 함수이다.

33~34행에서 1~9를 뒤에 오는 숫자로 추가하여 자릿수를 늘려간다. 여기서 0은 미리 제외 시켰다. 이 방법은 수학적 탐색영역의 배제의 한 방법으로 0으로 끝나는 수는 짝수이므로 소수가 될 수 없으므로 미리 제외시켰다.

백트래킹 함수의 의미는 다음과 같다.

f(num, len) = 현재 숫자 num과 그 길이 len을 의미

메인 함수에서 f(0, 0)으로 호출하면, 다음 구조로 호출이 이루어진다.



이런 구조로 길이가 n인 순열이 생성되고, 생성된 수를 4~11행의 소수 판별함수로 오른쪽부터 하나씩 절단하면서 소수인지 판단한다. 이 때 n/10으로 수를 분리하면 된다. 만약 오른쪽을 절단하면서 체크하는 과정에서 하나라도 소수가 아니면 바로 취소하고, 다음 숫자로 넘어간다. 만약 오른편 절단 가능 소수임이 판단되면 전체 개수를 저장하는 변수 cnt 값을 1증가시키고, 그 수를 화면에 바로 출력한다.

이 함수의 계산량은 순열을 생성하는 부분에 $O(9^n)$이 되고, 생성된 길이가 n인 각 수 x에 대해 $O(n\sqrt{x})$의 시간이 걸리므로, 전체 계산량은 $O(9^n \cdot n\sqrt{x})$이다. n이 6이상만 되어도 속도가 점점 느려짐을 알 수 있다.

이보다 더 탐색영역을 배제할 수 있는 부분을 생각해보자. 앞에서 0을 배제시켰듯이 수학적 배제 방법을 생각해보자.

1. 한 자릿수 중 소수는 2, 3, 5, 7 밖에 없다. 따라서 제일 높은 자릿수의 값은 2, 3, 5, 7만 될 수 있다. (∵오른편 절단 가능 소수이므로)
2. 두 자릿수 이상 넘어가면서 마지막 자릿수 값은 짝수가 될 수 없고(∵2의 배수), 마찬가지로 5도 될 수 없다(∵5의 배수). 따라서 남는 숫자는 1, 3, 7, 9만 남게 된다. 10개의 자릿수를 모두 다 탐색하는 것에 비해 가짓수가 현저히 줄어들게 됨으로 탐색이 빨라진다. (가지치기)
3. 자릿값을 늘려갈 때 현재 숫자가 소수인지 판별하여 소수인 숫자에만 뒤에 1, 3, 7, 9를 붙여가며 늘려간다. 숫자가 커지면 소수 판별에도 시간이 많이 걸리므로 시간을 줄이기 위해 $O(\sqrt{n})$ 소수 판별 알고리즘을 사용한다.

이 배제 방법을 토대로 소스를 개선시켜보자.

줄	코드	참고
1	`#include<stdio.h>`	26: 두 번째 자릿수부터는 1, 3, 7, 9 39: 시작 수는 2, 3, 5, 7
2		
3	`int n, cnt;`	
4		
5	`int isprime(int x)`	
6	`{`	
7	`  for(int i=2; i*i<=x; i++)`	
8	`    if(x%i==0)`	
9	`      return 0;`	
10	`    return 1;`	
11	`}`	
12		

```c
void f(int num, int len)
{
  if(len==n)
  {
    if(isprime(num))
    {
      cnt++;
      printf("%d\n", num);
    }
    return ;
  }
  else
  {
      if(isprime(num))
      {
         f(num*10+1, len+1);
         f(num*10+3, len+1);
         f(num*10+7, len+1);
         f(num*10+9, len+1);
      }
   }
}

int main()
{
  scanf("%d", &n);
  f(2,1); f(3,1); f(5,1); f(7,1);
  printf("%d", cnt);
}
```

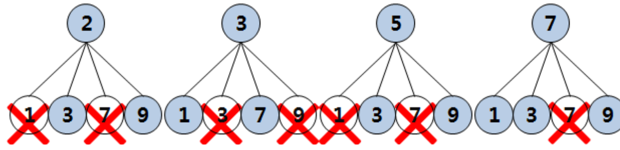
이 배제 방법을 토대로 n이 1, 2, 3일 때 결과를 살펴보자.

〈 N의 크기에 따른 상태 공간 〉

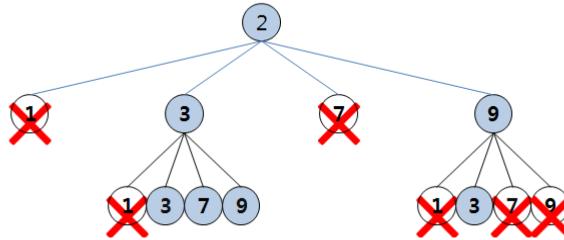
n = 1일 때,

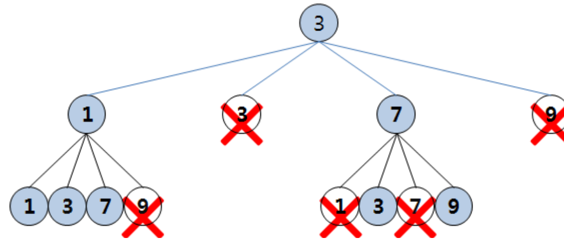


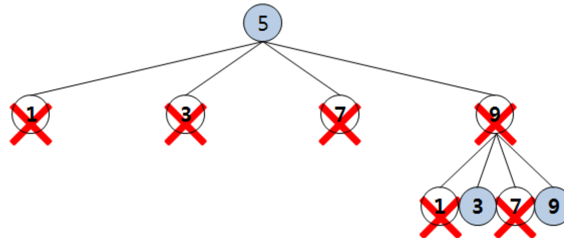
n = 2일 때,

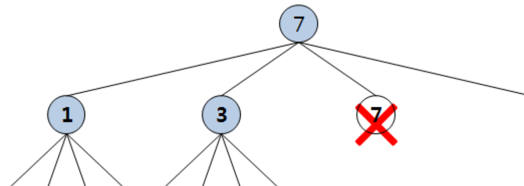


n = 3일 때,









이 소스의 계산량은 $O(4^n\sqrt{x})$ 이며 가지치기 전략에 의해 실질적으로는 수행시간을 더욱 단축할 수 있다.

문제 10

minimum sum(S)

n*n개의 수가 주어진다. (1<=n<=10)

이때 겹치지 않는 각 열과 각 행에서 수를 하나씩 뽑는다.
(즉, 총 n개의 수를 뽑을 것이다, 그리고 각 수는 100 이하의 값이다.)

이 n개의 수의 합을 구할 때 최솟값을 구하시오.

입력

첫 줄에 n이 입력된다. 다음 줄부터 n+1줄까지 n개씩의 정수가 입력된다.

출력

구한 최소 합을 출력한다.

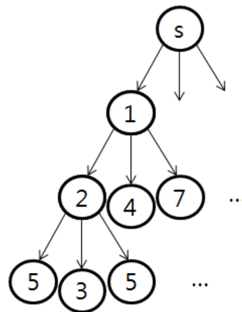
입력 예	출력 예
3 1 5 3 2 4 7 5 3 5	7

풀이

먼저 주어진 입력예제에 대해서 전체탐색법으로 접근하는 방법에 대해서 알아보자.

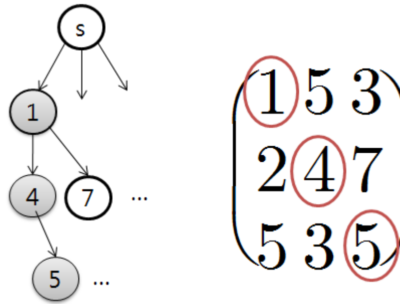
$$\begin{pmatrix} 1 & 5 & 3 \\ 2 & 4 & 7 \\ 5 & 3 & 5 \end{pmatrix}$$

행과 열이 중복되지 않아야 하므로 일단 각 행에서 1개 이상의 값은 얻을 수 없다. 그리고 서로 같은 열도 중복되지 않아야 하므로 위 문제는 다음과 같이 구조화할 수 있다.



탐색 구조

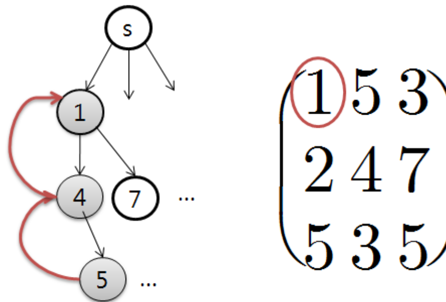
위 그림은 1행 1열의 1을 선택했을 때의 탐색 구조의 일부를 나타낸다. 1행에서 1열을 택했으므로 2행의 {2, 4, 7}중 1열의 {2}는 선택할 수 없다. 따라서 깊이우선으로 구할 수 있는 첫 번째 해는 아래 그림과 같다.



처음으로 구한 해

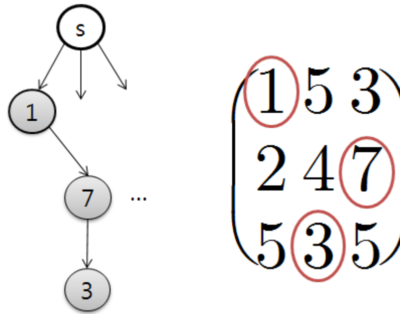
처음으로 구한 해는 위와 같이 1-4-5를 탐색하여 구한 10이다. 문제의 조건에 따라 1행 1열, 2행 2열을 선택했기 때문에 3행에서는 1열과 2열을 선택할 수 없다. 이 해 10은 최종적인 해인지 아닌지 현재 상태로는 확정할 수 없다. 따라서 백트래킹하여 가능한 모든 해를 구해야만 최종적으로 해를 구할 수 있다.

위의 상태에서 백트래킹 하면 다음과 같은 상태가 된다.



백트랙 후의 상태

위의 그림과 같은 상태에서는 7을 선택하여 계속해서 깊이우선탐색을 진행할 수 있다.



2번째로 구한 해 11

위의 그림에서 다시 11이라는 해를 구할 수 있다. 여기서 다시 백트랙하여 구할 수 있는 모든 해를 나열하면 다음과 같다.

$$(5,2,5)=12,\ (5,7,5)=17,\ (3,2,3)=8,\ (3,4,5)=12$$

따라서 이 문제에서 구할 수 있는 최소 점수는 3, 2, 3을 선택하여 얻을 수 있는 8점이 된다. 이 방법으로 작성한 소스코드는 다음과 같다.

코드	참고

```
#include<stdio.h>
int m[11][11];
int col_check[11];
int n, min_sol=0x7fffffff;

void input(void)
{
  scanf("%d", &n);
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      scanf("%d", &m[i][j]);
}

void solve(int row, int score)
{
  if(row==n)
  {
    if(score<min_sol)
      min_sol = score;
    return;
  }
  for(int i=0; i<n; i++)
  {
    if(col_check[i]==0)
    {
      col_check[i]=1;
      solve(row+1, score+m[row][i]);
      col_check[i]=0;
    }
  }
  return;
}

int main()
{
  input();
  solve(0, 0);
  printf("%d", min_sol);
  return 0;
}
```

문제 11

앱(S)

우리는 스마트폰을 사용하면서 여러 가지 앱 (App)을 실행하게 된다. 대개의 경우 화면에 보이는 '실행중'인 앱은 하나뿐이지만 보이지 않는 상태로 많은 앱이 '활성화' 되어 있다. 앱들이 활성화되어 있다는 것은 화면에 보이지 않더라도 메인메모리에 직전의 상태가 기록되어 있는 것을 말한다. 현재 실행중이 아니더라도 이렇게 메모리에 남겨두는 이유는 사용자가 이전에 실행하던 앱을 다시 불러올 때에 직전의 상태를 메인메모리로부터 읽어 들여 실행 준비를 빠르게 마치기 위해서이다.

하지만 스마트폰의 메모리는 제한적이기 때문에 한 번이라도 실행했던 모든 앱을 활성화된 채로 메인메모리에 남겨두다 보면 메모리 부족 상태가 되기 쉽다. 새로운 앱을 실행시키기 위해 필요한 메모리가 부족해지면 스마트폰의 운영체제는 활성화되어 있는 앱들 중 몇 개를 선택하여 메모리로부터 삭제하는 수밖에 없다. 이러한 과정을 앱의 '비활성화'라고 한다.

메모리 부족 상황에서 활성화되어있는 앱들을 무작위로 필요한 메모리만큼 비활성화하는 것은 좋은 방법이 아니다. 비활성화된 앱들을 재실행할 경우 그만큼 시간이 더 필요하기 때문이다. 여러분은 이러한 앱의 비활성화 문제를 스마트하게 해결하기 위한 프로그램을 작성해야 한다.

현재 n개의 앱,A1, .. , An 이 활성화되어 있다고 가정하자. 이들 앱Ai 는 각각 mi 바이트만큼의 메모리를 사용하고 있다. 또한, 앱 Ai 를 비활성화한 후에 다시 실행하고자 할 경우, 추가적으로 들어가는 비용(시간 등)을 수치화한 것을 ci라고 하자. 이러한 상황에서 사용자가 새로운 앱 B를 실행하고자 하여, 추가로 M 바이트의 메모리가 필요하다고 하자. 즉, 현재 활성화되어 있는 앱 A1,..,An 중에서 몇 개를 비활성화하여 M 바이트 이상의 메모리를 추가로 확보해야 하는 것이다. 여러분은 그 중에서 비활성화했을 경우의 비용 ci의 합을 최소화하여 필요한 메모리 M 바이트를 확보하는 방법을 찾아야 한다.

앱(S) (계속)

입력

첫 줄에는 정수 n과 M이 공백문자로 구분되어 주어지며,

둘째 줄과 셋째 줄에는 각각 n개의 정수가 공백문자로 구분되어 주어진다.

둘째 줄의 n개의 정수는 현재 활성화되어 있는 앱 A1,..., An이 사용 중인 메모리의 바이트 수인 m1, ..., mn을 의미하며,

셋째 줄의 n 개의 정수는 각 앱을 비활성화했을 경우의 비용 c1,..., cn을 의미한다.

[입력의 정의역]

1 ≤ n ≤ 100

1 ≤ M ≤ 10,000,000

1 ≤ m1, ..., mn ≤ 10,000,000

0 ≤ c1,..., cn≤ 100

출력

필요한 메모리 M 바이트를 확보하기 위한 앱 비활성화의 최소의 비용을 계산하여 한 줄에 출력해야 한다.

입력 예	출력 예
5 60 30 10 20 35 40 3 0 3 5 4	6

출처: 한국정보올림피아드(2013 지역본선 중고등부)

풀이

이 문제는 새로운 앱을 실행하기 위해 활성화되어 있는 앱들 중 몇 개를 비활성화해서 메모리 M 이상을 확보하는 데 드는 비용을 최소화하는 문제이다.

이 상황을 잘 생각해보면 배낭 문제(문제 17)와 유사해 보인다.

	배낭 문제		앱
N	배낭에 담을 수 있는 물건 개수	n	비활성화 할 수 있는 앱의 개수
W	배낭의 무게	M	확보해야할 메모리량
Wi	각 물건의 무게	mi	각 앱의 메모리 사용량
Vi	물건의 가치	ci	앱의 비활성화에 드는 비용

배낭 문제에서는 배낭의 무게 W를 넘지 않으면서 물건 가치를 최대로 높이는 경우를 찾는 것이고, 앱 문제에서는 메모리를 M 이상을 확보하면서 비활성화에 드는 최소 비용을 찾는 것이다.

문제에서 제시한 상황은 다음 표와 같다.

앱의 번호(i)	사용 중인 메모리(mi)	비활성화 비용(ci)
1	30	3
2	10	0
3	20	3
4	35	5
5	40	4

요구하는 메모리가 60이므로 비활성화 비용을 최소화하면서 60 이상의 메모리를 확보하기 위해서는 1, 2, 3번의 앱을 비활성화시켜야 한다.(30+10+20=60, 3+0+3=6)

배낭 문제와 유사하기 때문에 배낭 문제에서 사용한 알고리즘을 이 문제에 맞게 변형시켜보자. 배낭문제에서는 f(1, 0)을 호출해서 답을 구했지만, 이번에는 다른 방법으로 설계해도 똑같은 결과가 나온다는 것을 보여주기 위해 반대로 f(n, M)으로 설계하였다.

줄	코드	참고
1	`#include <stdio.h>`	8: 앱 번호 i, 남은 메모리 r
2	`#define MAXV 999999`	
3		
4	`int M, n, i, m[101], c[101];`	
5		
6	`int min(int a, int b) { return a<b ? a:b;}`	
7		
8	`int f(int i, int r)`	
9	`{`	
10	`  if(i==0)`	
11	`  {`	
12	`    if(r<=0) return 0;`	
13	`    else return MAXV;`	
14	`  }`	
15	`  else if (r<0)`	
16	`    return f(i-1, r);`	
17	`  else`	
18	`    return min(f(i-1,r), f(i-1,r-m[i])+c[i]);`	
19	`}`	
20		
21	`int main()`	
22	`{`	
23	`  scanf("%d %d", &n, &M);`	
24	`  for(i=1; i<=n; i++) scanf("%d", &m[i]);`	
25	`  for(i=1; i<=n; i++) scanf("%d", &c[i]);`	
26	`    printf("%d", f(n, M));`	
27	`  return 0;`	
28	`}`	

함수 f의 의미는 다음과 같다.

f(i, r) = 1~i번째 앱까지 고려했을 때, 메모리 r이상 확보하기 위한 최소 비용

설계 방법의 변경과 최솟값을 구하는 부분에서 약간의 소스가 변경되었다. 이 방법으로는 이 문제를 완벽하게 해결하기에는 시간이 부족하다. 고급편에서 이 알고리즘의 시간을 줄이는 방법에 대해서 다룬다.

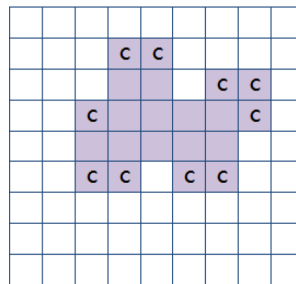
문제 12

치즈

n*m(5≤ n, m ≤100)의 모눈종이 위에 아주 얇은 치즈가 [그림 1]과 같이 표시되어 있다. 단, n은 세로 격자의 수이고, m은 가로 격자의 수이다. 이 치즈는 냉동 보관을 해야만 하는데 실내온도에 내어놓으면 공기와 접촉하여 천천히 녹는다.

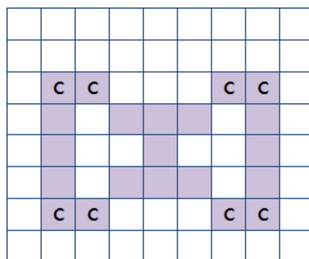
그런데 이러한 모눈종이 모양의 치즈에서 각 치즈 격자(작은 정사각형 모양)의 네 변 중에서 적어도 두 변 이상이 실내온도의 공기와 접촉한 것은 정확히 한 시간 만에 녹아 없어져 버린다.

따라서 아래 [그림 1] 모양과 같은 치즈(회색으로 표시된 부분)라면 C로 표시된 모든 치즈 격자는 한 시간 후에 사라진다.

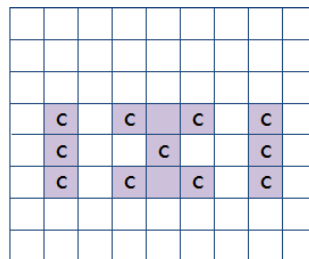


[그림 1]

[그림 2]와 같이 치즈 내부에 있는 공간은 치즈 외부 공기와 접촉하지 않는 것으로 가정한다. 그러므로 이 공간에 접촉한 치즈 격자는 녹지 않고 C로 표시된 치즈 격자만 사라진다. 그러나 한 시간 후, 이 공간으로 외부공기가 유입되면 [그림 3]에서와 같이 C로 표시된 치즈 격자들이 사라지게 된다.



[그림 2]



[그림 3]

치즈 (계속)

모눈종이의 맨 가장자리에는 치즈가 놓이지 않는 것으로 가정한다. 입력으로 주어진 치즈가 모두 녹아 없어지는 데 걸리는 정확한 시간을 구하는 프로그램을 작성하시오.

입력

첫째 줄에는 모눈종이의 크기를 나타내는 두 개의 정수 n, m(5 ≤ n, m ≤100)이 주어진다. 그 다음 n개의 줄에는 모눈종이 위의 격자에 치즈가 있는 부분은 1로 표시되고, 치즈가 없는 부분은 0으로 표시된다. 또한, 각 0과 1은 하나의 공백으로 분리되어 있다.

출력

출력으로는 주어진 치즈가 모두 녹아 없어지는 데 걸리는 정확한 시간을 정수로 첫 줄에 출력한다.

입력 예	출력 예
8 9 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 1 0 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 0 0 0 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	4

출처: 한국정보올림피아드(2000 지역본선 초등부)

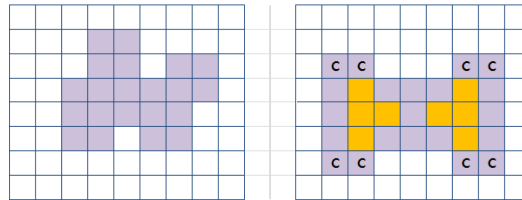
풀이

이 문제는 앞에서 다루었던 깊이우선탐색이나 너비우선탐색을 이용한 flood fill 기법과 다양한 문제해결 기법을 응용해야 되는 문제이다. 이 문제를 통해서 많은 것을 배울 수 있다.

치즈가 녹는 규칙과 치즈의 초기 상태가 주어졌을 때, 치즈가 다 녹는 데 걸리는 시간을 출력하는 문제이다.

시간이 진행되면서 치즈가 어떻게 녹는지를 시뮬레이션하는 방법으로 풀 수 있다.

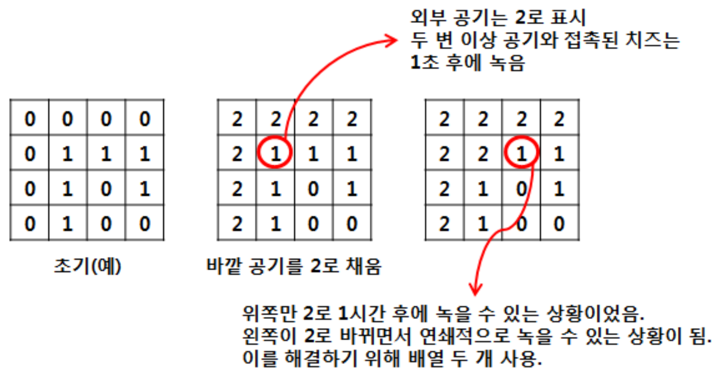
즉, 매 시간마다 치즈의 상태가 변해가는 모습을 기록한 후, 최종 상태까지 걸린 시간을 출력하면 된다.



주황 격자에 해당하는 내부 공간은 치즈를 녹이지 못하기 때문에 한 시간 후에는 C로 표시된 치즈만 사라진다.

바깥쪽 공기와 치즈의 안쪽 구멍을 구분하는 것이 이 문제의 핵심이다. 먼저 가장 쉽게 모든 것을 구현하며 풀어보는 방법을 알아보자. 기본 아이디어는 1시간마다 백트래킹으로 바깥 공기를 다시 체크하는 방법이다.

단, 녹을 치즈를 바로 공기로 바꾸면 영향을 받기 때문에 배열 두 개를 사용한다.



위 아이디어를 flood fill기법으로 해결한 소스코드는 다음과 같다.

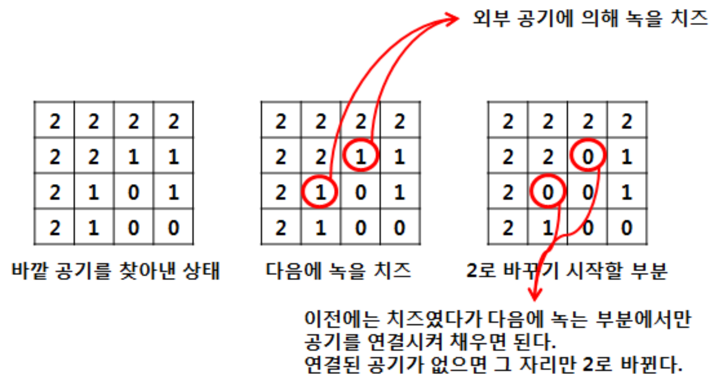
코드 / 참고

```
#include <stdio.h>
int a1[101][101], a2[101][101];
int n, m;
void copy()
{
  int i, j;
  for(i=1; i<=n; i++)
    for(j=1; j<=m; j++)
      a1[i][j]=a2[i][j];
}
void fill1(int x, int y)
{
  if(x<1 || y<1 || x>n || y>m) return;
  if(a1[x][y]==0)
  {
    a1[x][y]=2;
    fill1(x+1,y);
    fill1(x-1,y);
    fill1(x,y+1);
    fill1(x,y-1);
  }
}
int check(int x, int y)
{
  int t=0;
  if(a1[x+1][y]==2) t++;
  if(a1[x-1][y]==2) t++;
  if(a1[x][y+1]==2) t++;
  if(a1[x][y-1]==2) t++;
  return t;
}
int main()
{
  int i, j, hour=0, count;
  scanf("%d %d",&n, &m);
  for(i=1; i<=n; i++)
    for(j=1; j<=m; j++)
    {
      scanf("%d",&a1[i][j]);
      a2[i][j]=a1[i][j];
```

```
    }
  while(1)
  {
    fill1(1,1);
    count=0;
    for(i=1; i<=n; i++)
      for(j=1; j<=m; j++)
      {
        if(a1[i][j]==1 && check(i,j)>=2)
        {
          a2[i][j]=0;
          count++;
        }
      }
    if(count==0)
    {
      printf("%d", hour);
      break;
    }
    hour++;
    copy();
  }
  return 0;
}
```

위 알고리즘에서 조금 더 개선하는 방법을 생각해보자. 한 시간마다 바깥 공기 덩어리를 다시 체크하지 않고, 녹았을 때만 그 자리에서 다시 연결된 공기를 체크한다.

이렇게 하면 조금 더 빠르게 해결할 수 있다.



위 알고리즘으로 fill을 fill1과 fill2로 만들 수 있다.

```
void fill1(int x, int y)
{
  if(x<1 || y<1 || x>n || y>m) return;
  if(a1[x][y]==0)
  {
    a1[x][y]=2;
    fill1(x+1,y);
    fill1(x-1,y);
    fill1(x,y+1);
    fill1(x,y-1);
  }
}

void fill2(int x, int y)
{
  if(x<1 || y<1 || x>n || y>m) return;
  if(a2[x][y]==0)
  {
    a2[x][y]=2;
    fill2(x+1,y);
    fill2(x-1,y);
    fill2(x,y+1);
    fill2(x,y-1);
  }
}
```

이와 같이 작성하면 main()은 다음과 같이 수정된다.

```
int main()
{
  int i, j, hour=0, count;
  scanf("%d %d",&n, &m);
  for(i=1; i<=n; i++)
    for(j=1; j<=m; j++)
    {
      scanf("%d",&a1[i][j]);
      a2[i][j]=a1[i][j];
    }
  fill1(1,1);
```

```
  fill2(1,1);
  while(1)
  {
    count=0;
    for(i=1; i<=n; i++)
      for(j=1; j<=m; j++)
      {
        if(a1[i][j]==1 && check(i,j)>=2)
        {
          a2[i][j]=0;
          count++;
        }
      }
    if(count==0)
    {
      printf("%d", hour); break;
    }
    for(i=1; i<=n; i++)
      for(j=1; j<=m; j++)
        if(a1[i][j]==1 && a2[i][j]==0)
          fill2(i,j);
    hour++;
    copy();
  }
  return 0;
}
```

위의 알고리즘들을 조금 더 개선해 보자. 바뀔 부분을 찾기 위해 모두 검색하는 것이 아니라, 저장해 두었다가 처리한다.



```
void fill3(int x, int y)
{
  if(x<1 || y<1 || x>n || y>m) return;
  if(a1[x][y]==3 || a1[x][y]==0)
  {
    a1[x][y]=2;
    fill3(x+1,y);
    fill3(x-1,y);
    fill3(x,y+1);
    fill3(x,y-1);
  }
}

int main()
{
  int i, j, hour=0, count;
  scanf("%d %d",&n, &m);
  for(i=1; i<=n; i++)
    for(j=1; j<=m; j++)
    {
      scanf("%d",&a1[i][j]);
      a2[i][j]=a1[i][j];
    }
  fill3(1,1);
  while(1)
  {
    count=0;
    for(i=1; i<=n; i++)
      for(j=1; j<=m; j++)
      {
        if(a1[i][j]==1 && check(i,j)>=2)
        {
          a2[i][j]=0;
          count++;
        }
      }
    if(count==0)
    {
      printf("%d", hour); break;
    }
```

```
    for(i=1; i<=n; i++)
      for(j=1; j<=m; j++)
        if(a1[i][j]==1 && a2[i][j]==0)
          fill2(i,j);
    hour++;
    copy();
  }
  return 0;
}
```

마지막으로 위 아이디어를 조금 더 효율적으로 접근한 소스코드를 소개한다. 이 소스코드에는 지금까지 배웠던 다양한 기법들이 적용되었기 때문에, 자세히 분석해서 익힐 수 있도록 한다.

```
#include <cstdio>
int dx[4]={1,0,-1,0}, dy[4]={0,1,0,-1}, h, w, S[110][110],
res;
bool inside(int a, int b)
{
  return ((0<=a && a<h) && (0<=b && b<w));
}
bool done(void)
{
  int cnt=0;
  for(int i=0; i<h; i++) for(int j=0; j<w; j++)
    if(S[i][j]==-1 || S[i][j]>2) S[i][j]=0;
    else if(S[i][j]==2 || S[i][j]==1)
      S[i][j]=1, cnt++;
  return cnt==0;
}
int solve(int a, int b)
{
  S[a][b]=-1;
  for(int i=0; i<4; i++)
    if(inside(a+dx[i], b+dy[i]))
    {
      if(S[a+dx[i]][b+dy[i]]==0)
```

줄	코드	참고

```
        solve(a+dx[i],b+dy[i]);
      else if(S[a+dx[i]][b+dy[i]]>0)
        S[a+dx[i]][b+dy[i]]++;
    }
}
int main()
{
  scanf("%d %d",&h,&w);
  for(int i=0; i<h; i++) for(int j=0; j<w; j++)
    scanf("%d", &S[i][j]);
  for(res=0; !done(); res++) solve(0,0);
  printf("%d", res);
  return 0;
}
```

문제 13

두 색 칠하기 (bicoloring)

평면 위에 지도가 있을 때, 각 영역을 인접한 다른 영역과 구분할 수 있게 서로 다른 색으로 칠하고자 한다면, 네 가지 색만 있으면 된다는 4색 정리라는 것이 있다. 이 정리는 100년이 넘게 증명되지 않은 채로 남아 있다가 1976년에서야 컴퓨터의 도움을 받아서 증명될 수 있었다.

이 문제는 그래프의 정점을 칠하는 문제로 구조화하여 풀 수 있다. 어떤 연결 그래프가 주어졌을 때 그 그래프를 두 색으로 칠할 수 있는지, 즉 모든 정점을 빨간색 또는 검은색으로 칠할 때 인접한 정점이 같은 색으로 칠해지지 않게 할 수 있는지 알아보자.

문제를 단순하게 하기 위해 그래프가 연결 그래프이고 무방향 그래프이며 자체 루프가 없다고 가정하자. 0부터 n-1까지의 n개의 정점과 간선의 수 m이 입력될 때, 2 가지 색깔로 칠할 수 있는지 결정하는 프로그램을 작성하시오.

입력

첫째 줄에는 정점의 개수 n(1 ≤ n ≤ 200)과 간선의 수 m이 입력된다.

둘째 줄부터 m줄에 걸쳐서 각 간선이 연결하는 정점의 번호가 공백으로 구분되어 입력된다.

출력

입력된 그래프가 두 색으로 칠할 수 있는 그래프인지를 판단하고 아래 예에 나온 형식에 맞게 결과를 출력하라.

입력 예	출력 예
3 3 0 1 1 2 2 0	IMPOSSIBLE
9 8 0 1 0 2 0 3 0 4 0 5 0 6 0 7 0 8	OK

풀이

그래프이기 때문에 비선형 전체탐색으로 해결할 수 있다.

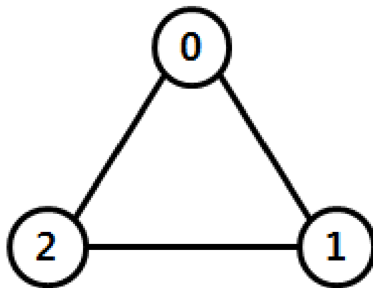
임의의 정점에 어떤 색이 칠해졌는지는 관계없다. 주변의 정점들과의 색깔만 다르면 되기 때문에 시작 정점을 검은색, 빨간색 중 아무거나 칠해도 관계없다.

시작 정점에서 임의의 색깔로 출발하여 인접한 정점에는 다른 색깔을 칠하도록 하자. 이 과정에서 깊이우선탐색이나 너비우선탐색 등의 방법은 모두 가능하다.

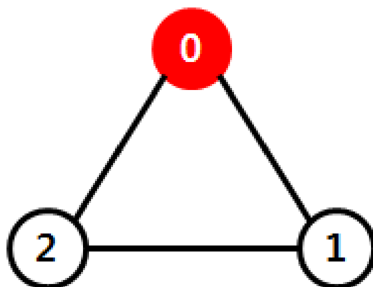
여기서는 깊이우선탐색을 기반으로 한 다음 알고리즘으로 색깔을 칠해보자. 1은 검은색, 2는 빨간색을 의미한다.

1. 정점 0을 1로 칠하고 정점 0에 연결된 임의의 정점으로 탐색을 진행.
2. 현재 정점을 1로 칠해보고 불가능하면 현재 정점을 2로 칠해보고 탐색을 진행.
3. 모든 정점에 색깔을 칠할 수 있으면 OK, 아니면 IMPOSSIBLE.

입력 예시1 로 주어진 그래프를 칠해가는 과정은 다음과 같다.

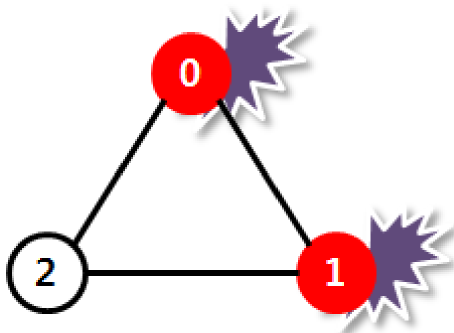


초기 그래프 상태
3개의 정점과, 3개의 간선을 가지는 그래프이다.

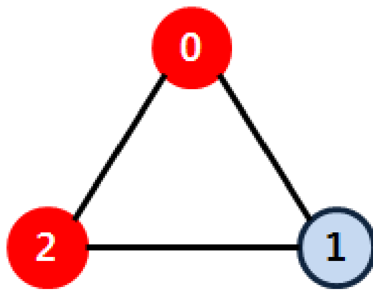


출발정점을 빨간색으로 칠하고 다음으로 1번 정점으로 탐색을 옮김

출발정점을 검은색으로 칠해도 의미는 같음.

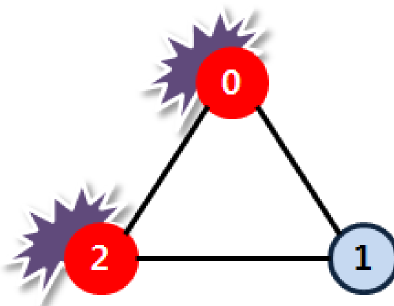


먼저 1번 정점을 빨간색으로 칠해본다. 하지만 정점 0과 같은 색깔이므로 사용 불가 백트랙!!

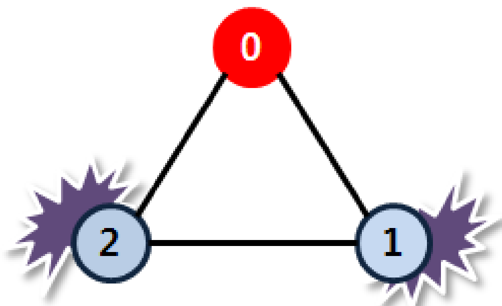


다음으로 검은색으로 칠해본다. 주변의 정점들과 색깔이 다르기 때문에 가능.

계속하여 2번 정점으로 탐색을 이동.



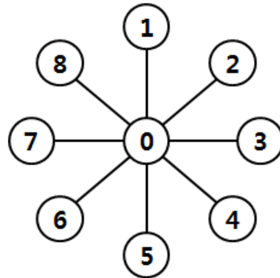
2번 정점을 먼저 빨간색으로 칠해본다. 하지만 0번 정점과 색깔이 같으므로 사용 불가 백트랙!!



2번 정점을 다시 검은색으로 칠해본다. 역시 1번 정점과 색깔이 같으므로 사용 불가!! 백트랙!!

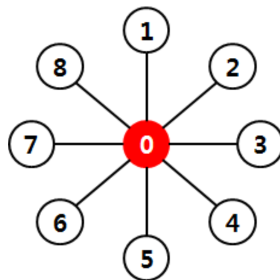
이와 같은 방법으로 끝까지 백트랙을 해도 채울 수 있는 방법이 없으므로 결과는 IMPOSSIBLE이 된다.

다음으로 2번째 예제의 경우를 살펴보자. 2번째 예제는 다음과 같은 방법으로 칠할 수 있으므로 처리할 수 있다.

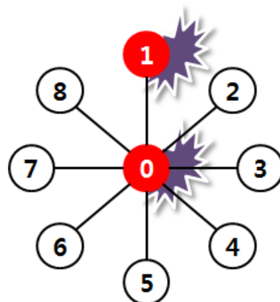


초기 상태는 다음과 같다.

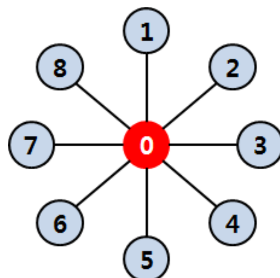
9개의 정점과 8개의 간선을 가지는 트리 형태의 그래프이다. (사실 트리는 항상 2가지 색깔로 칠할 수 있다. 이는 수학적으로 증명된다.)



출발 정점을 빨간색으로 칠하고 1번 정점으로 탐색을 이동한다.



1번을 빨간색으로 칠하면 0번과 색깔이 같으므로 불능!! 백트랙!!



검은 색으로 칠하면 이상 없음...

이와 같은 과정으로 나머지 모든 정점도 검은색으로 처리할 수 있으므로 2가지 색깔로 처리할 수 있다.

위의 과정을 작성한 소스코드는 다음과 같다.

코드	참고

```
#include <stdio.h>

int n, m, G[200][200], visited[200];

void solve(int v, int c)
{
  visited[v]=c;
  int can=1;
  for(int i=0; i<n; i++)
    if(G[v][i] && visited[i]==c) can=0;

 if(!can)
  {
    visited[v]=0;
    return;
  }
  for(int i=0; i<n; i++)
  {
    if(!visited[i] && G[v][i])
    {
      solve(i, 1);
      solve(i, 2);
    }
  }
}

int main()
{
  scanf("%d %d", &n, &m);
  for(int i=0; i<m; i++)
  {
    int s, e;
    scanf("%d%d",&s,&e);
    G[s][e]=G[e][s]=1;
  }
  solve(0, 1);
  for(int i=0; i<n; i++)
    if(visited[i]==0)
    {
```

```
      puts("IMPOSSIBLE");
      return 0;
    }
  printf("OK");
  return 0;
}
```

위 알고리즘은 그래프를 인접행렬로 표현한 것이다. 이를 인접리스트로 표현하면 속도가 더 빨라진다. 이 문제의 경우에는 정점의 수가 적어서 인접행렬로도 해결이 되지만 인접리스트로도 작성하는 연습을 하는 것이 좋다.

인접리스트로 작성한 소스코드는 다음과 같다. 실전에서는 인접리스트 표현이 더 자주 활용되므로 익혀두기 바란다.

```
#include <stdio.h>
#include <vector>

int n, m, visited[200];
std::vector<int> G[200];

void solve(int v, int c)
{
  visited[v]=c;
  int can=1;
  for(int i=0; i<G[v].size(); i++)
    if(visited[G[v][i]]==c) can=0;
  if(!can)
  {
    visited[v]=0;
    return;
  }
  for(int i=0; i<G[v].size(); i++)
  {
    if(!visited[G[v][i]])
    {
      solve(G[v][i], 1);
      solve(G[v][i], 2);
    }
```

```
  }
}

int main()
{
  scanf("%d %d", &n, &m);
  for(int i=0; i<m; i++)
  {
     int s, e;
     scanf("%d %d",&s,&e);
     G[s].push_back(e);
     G[e].push_back(s);
  }
  solve(0, 1);
  for(int i=0; i<n; i++)
   if(visited[i]==0)
   {
     puts("IMPOSSIBLE");
     return 0;
   }
   printf("OK");
   return 0;
}
```

빨간색으로 표시된 코드는 인접행렬로 구현했을 때와의 차이가 나는 부분을 의미한다.

문제 14

maximum sum(S)

n개의 원소로 이루어진 집합이 있다. 이 집합에서 최대로 가능한 부분합을 구하는 것이 문제이다.

부분합이란 n개의 원소 중 i번째 원소로부터 j번째 원소까지의 연속적인 합을 의미한다(단, 1 < i <= j <= n). 만약 다음과 같이 6개의 원소로 이루어진 집합이 있다고 가정하자.

6 -7 3 -1 5 2

이 집합에서 만들어지는 부분합 중 최댓값은 3번째 원소부터 6번째 원소까지의 합인 9이다.

입력

첫 줄에 원소의 수를 의미하는 정수 n이 입력되고, 둘째 줄에 n개의 정수가 공백으로 구분되어 입력된다.
(단, 2 <= n <= 100 , 각 원소의 크기는 -1000부터 1000 사이의 정수이다.)

출력

주어진 집합에서 얻을 수 있는 최대 부분합을 출력한다.

입력 예	출력 예
6 6 -7 3 -1 5 2	9

풀이

이 문제는 n의 값이 클 경우에는 고민해야 할 부분 많은데 이 경우는 n의 최댓값이 100이기 때문에 $O(n^3)$으로도 처리할 수 있다. 따라서 단순히 선형으로 전체탐색하면 해를 구할 수 있다.

기본적인 알고리즘은 다음과 같다.

1. 구간합을 구할 구간의 시작점 s를 정한다(모든 값에 대하여).
2. 구간합을 구할 구간의 끝점 e를 정한다(모든 값에 대하여).
3. [s, e] 구간의 합을 구한다. 만약 지금까지 구한 합보다 더 크면 갱신한다.
4. 마지막까지 진행하고, 가장 큰 합을 출력한다.

먼저 각 구간의 시작과 끝을 구하기 위하여 다음과 같은 코드를 작성할 수 있다. 단, 구간의 끝은 시작보다 같거나 커야 한다.

```
int count=1;
for(int s=0; s<n; s++)
{
   for(int e=s; e<n; e++)
   {
      printf("[%d~%d]   ", s, e);
      if(count%7==0) puts("");
      count++;
   }
}
```

위의 코드는 모든 구간을 설정할 수 있는 코드이다. 따라서 $O(n^2)$에 모든 구간을 설정할 수 있다. 각 구간을 출력한 결과는 다음과 같다.

```
[0~0]  [0~1]  [0~2]  [0~3]  [0~4]  [0~5]  [1~1]
[1~2]  [1~3]  [1~4]  [1~5]  [2~2]  [2~3]  [2~4]
[2~5]  [3~3]  [3~4]  [3~5]  [4~4]  [4~5]  [5~5]
```

위 코드에서 정한 구간의 합을 정하면 된다. 이를 정리한 소스코드는 다음과 같다.

```c
#include <stdio.h>

int n, A[110], ans;

int main()
{
  scanf("%d", &n);
  for(int i=0; i<n; i++)
    scanf("%d", A+i);
  for(int s=0; s<n; s++)
  {
    for(int e=s, sum; e<n; e++)
    {
      sum=0;
      for(int k=s; k<=e; k++)
        sum+=A[k];
      ans=ans<sum ? sum:ans;
    }
  }
  printf("%d\n", ans);
  return 0;
}
```

이 문제는 아이디어에 따라서 훨씬 효율적인 방법들이 많이 있으므로, 다양한 생각들을 해보기 바란다.

문제 15

계단 오르기

길동이는 n개의 단으로 구성된 계단을 오르려고 한다.

길동이는 계단을 오를 때 기분에 따라서 한 번에 1단 또는 2단을 올라갈 수 있다.

계단의 크기 n이 주어질 때, 길동이가 이 계단을 올라갈 수 있는 모든 경우의 수를 구하는 프로그램을 작성하시오.

만약 계단이 3개라면 길동이는 1, 1, 1로 올라가는 법과 1, 2로 올라가는 법, 2, 1로 올라가는 법의 3가지 서로 다른 방법이 있다.

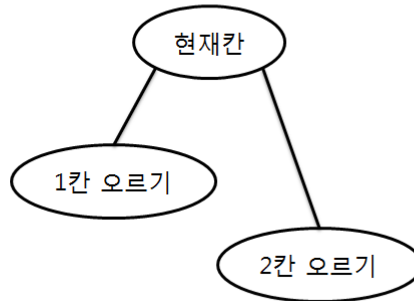
입력
계단의 수 n이 입력된다(단 n은 20보다 작은 자연수).

출력
길동이가 계단을 오르는 모든 방법의 수를 출력한다.

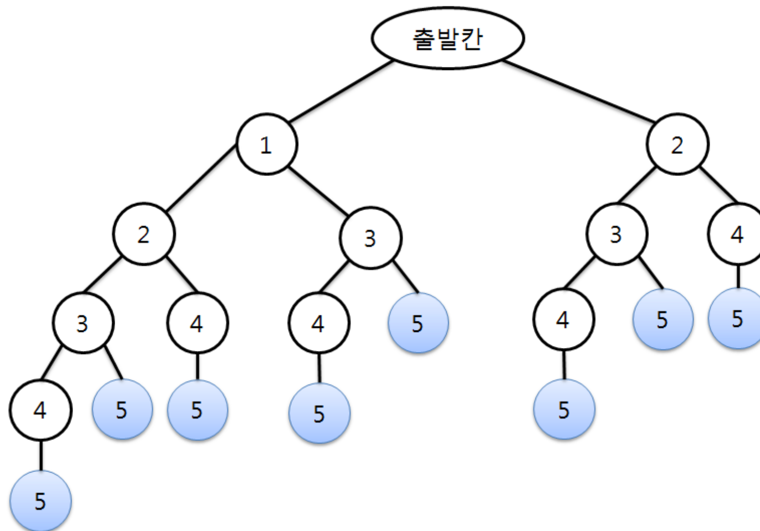
입력 예	출력 예
3	3

풀이

이 문제도 비선형구조로 전체탐색을 하여 해를 구할 수 있다. 현재 상태에서 1칸 또는 2칸을 올라갈 수 있으므로, 탐색구조를 다음과 같이 설정할 수 있다. 단, 주의할 점은 정확하게 n칸에 도착했을 때만 한 가지 경우로 처리해야한다는 점이다. 예를 들어 도착점까지 한 칸 남았을 경우에는 2칸을 올라갈 수 없다.



위의 탐색 과정으로 5칸의 계단을 오르는 과정을 보면 다음과 같다.



위와 같은 트리를 구성하면서 전체탐색을 하면 n은 5일 때, 방법은 8임을 알 수 있다. 이와 같은 구조의 탐색을 소스코드로 구현하면 다음과 같다.

줄	코드	참고
1	#include <stdio.h>	
2		
3	int n, ans;	
4		
5	void solve(int v)	
6	{	
7	if(v>n) return;	
8	if(v==n)	
9	{	
10	ans++;	
11	return;	
12	}	
13	solve(v+1);	
14	solve(v+2);	
15	}	
16		
17	int main()	
18	{	
19	scanf("%d", &n);	
20	solve(0);	
21	printf("%d\n", ans);	
22	}	

7행의 조건은 마지막 계단을 넘어가는 경우를 처리한다. 이 구문이 없으면 무한 재귀에 빠지게 된다.

문제 16

거스름 돈(S)

여러분은 실력을 인정받아 전 세계적으로 사용할 수 있는 자동판매기용 프로그램의 개발을 의뢰받았다. 거스름돈에 사용될 동전의 수를 최소화하는 것이다.

입력으로 거슬러 줘야할 돈의 액수와 그 나라에서 이용하는 동전의 가짓수 그리고 동전의 종류가 들어오면 여러 가지 방법들 중 가장 적은 동전의 수를 구하는 프로그램을 작성하시오.

입력

첫 번째 줄에는 거슬러 줘야할 돈의 액수 m이 입력된다.
(10 <= m <= 10,000)
다음 줄에는 그 나라에서 사용되는 동전의 종류의 수 n이 입력된다.
(1 <= n <= 10)
마지막 줄에는 동전의 수만큼의 동전 액수가 오름차순으로 입력된다.
(10 <= 액수 <= m)

출력

최소의 동전의 수를 출력한다.

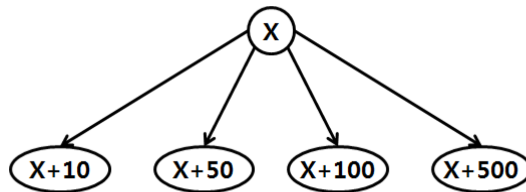
입력 예	출력 예
730 5 10 50 100 500 1250	6

풀이

이 문제는 매우 잘 알려진 유명한 문제 중 하나로 다양한 방법으로 해결할 수 있는 대표적인 문제이다. 이 단원에서는 전체탐색법을 기반으로 하여 해결하는 방법에 대해서 소개한다. 대부분의 문제들에서도 마찬가지지만 문제를 전체탐색으로 구조화하는 방법에 따라 해법의 계산량이 달라질 수 있다.

이 문제에서는 2가지 서로 다른 구조화로 해결하는 방법을 소개한다. 먼저 첫 번째 방법은 문제의 상태를 지금까지 지불한 액수로 설정하고, 서로 다른 동전 1개를 이용하여 지불하는 경우를 간선으로 생각할 수 있다.

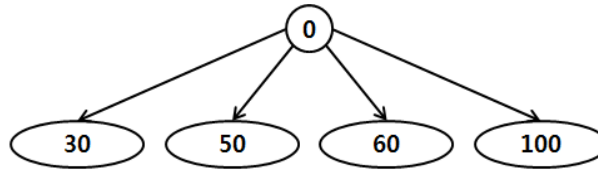
이 방법으로 구조화하는 방법은 다음 그림과 같다. 이 때 x의 값은 지금까지 지불한 액수이며 사용 가능한 동전은 4가지 종류로 10원, 50원, 100원, 500원일 때를 가정한 것이다.



서로 다른 1개의 지불하는 방법에 대해서 탐색 상태를 정의

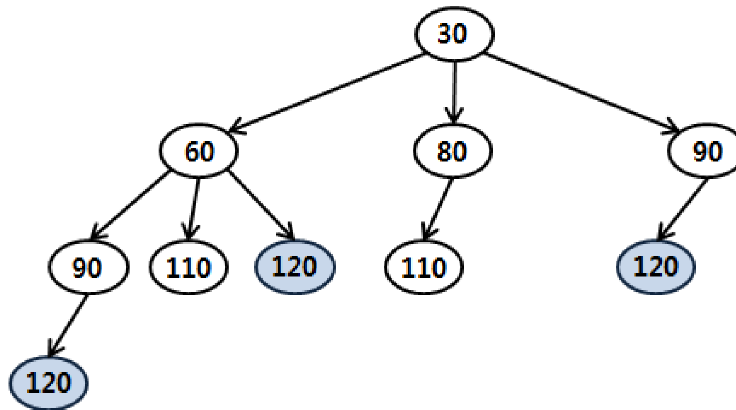
처음에 0원으로 출발하여 각 동전을 지불해 나가며, 지불할 금액과 일치할 때의 깊이가 지불한 동전의 개수이므로, 지불할 금액과 일치하는 최소 깊이를 구하는 문제가 된다. 만약 지불할 금액과 일치했거나 금액을 초과했을 경우에는 백트랙하면서 탐색을 진행하도록 코드를 작성하면 된다.

지불해야할 금액이 120원이고, 사용가능한 동전이 30원, 50원, 60원, 100원일 때의 전체 탐색구조는 다음과 같다. 먼저 처음 깊이 1까지의 구조이다.



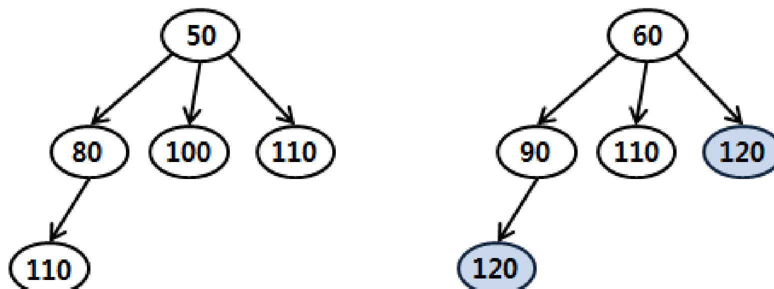
깊이 1까지의 탐색구조

다음은 30원 이하의 정점들의 전체적인 구조이다. 파란색 정점은 120원 지불에 성공한 것을 나타낸다.



30원 이하의 전체 탐색구조

계속해서 50원과 60원 정점의 전체 탐색구조를 나타낸다. 100원 이하에서는 더 이상의 탐색이 불가능하다.



나머지 모든 구조

```
#include<stdio.h>

int m, n, coin[10], ans=987654321;

void solve(int mon, int d)
{
  if(mon>m) return;
  if(mon==m)
  {
    if(d<ans) ans=d;
      return;
  }
  for(int i=0; i<n; i++)
    solve( mon+coin[i], d+1 );
}

int main()
{
  scanf("%d %d", &m, &n);
  for(int i=0; i<n ; i++ )
    scanf("%d", coin+i);
  solve(0, 0);
  printf("%d\n", ans);
  return 0;
}
```

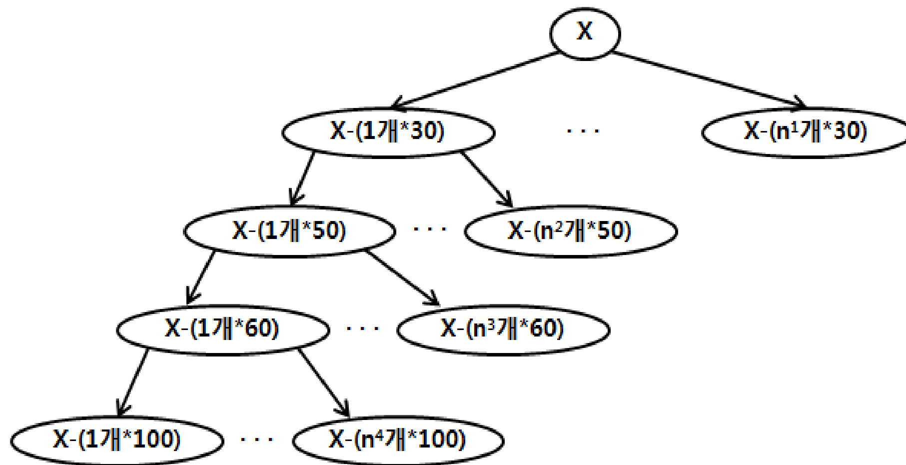
위 프로그램에서의 solve()함수는 다음과 같은 상태를 표현하고 있다.

solve(mon, d) = "d개의 동전으로 mon원을 사용한 상태"

이 방법은 정확하게 해를 구할 수는 있으나 이론상으로 최대 금액이 10,000원이고 최소 액수가 10원이므로 최대 깊이가 1,000까지 갈 수 있기 때문에 시간 내에 해결할 수 없다.

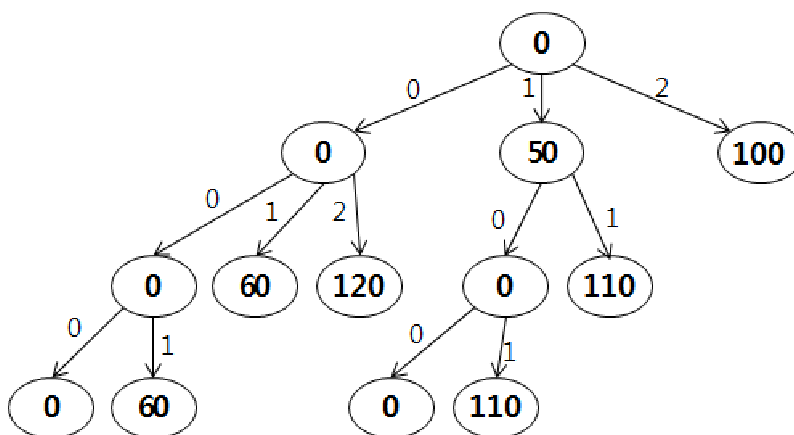
시간을 줄이기 위한 다양한 방법이 있지만, 이번에는 다른 구조를 이용하여 해결할 수 있는 방법을 소개한다. 탐색구조를 어떻게 설계하느냐에 따라서 해법의 계산량이 달라질 수 있다는 것을 알고, 문제를 해결할 때, 탐색구조를 어떻게 구성해야하는지 먼저 고민하는 것이 중요하다.

이번에 소개하는 구조는 이전과는 달리 같은 깊이에서는 같은 동전으로만 지불하는 방법으로 구조를 구성한다. 한 깊이에서 간선의 수는 해당 깊이의 동전을 0개로부터 해당 금액을 최대한 지불할 수 있는 최대한의 수로 설정하여 진행한다. 다음 그림을 통하여 자세히 알아보자. 동전은 10원, 50원, 100원, 500원이다.



깊이별로 다른 동전을 사용하도록 구성한 탐색구조

실제 지불할 금액은 120원, 지불가능한 동전의 수는 50원, 60원, 100원일 때의 경우 전체 구조는 다음과 같다.



깊이 별로 지불할 동전을 달리한 탐색 구조

이 방법으로 구현한 소스코드는 다음과 같다.

코드	참고

```
#include <stdio.h>

int n, m, coin[10], ans=987654321;

void solve(int mon, int k, int cnt)
{
  if(k==n || mon>m) return;
  if(mon==m)
  {
    if(ans>cnt) ans=cnt;
    return;
  }
  for(int i=0; mon+coin[k]*i<=m; i++)
    solve(mon+coin[k]*i, k+1, cnt+i);
}

int main()
{
  scanf("%d %d", &m, &n);
  for(int i=0; i<n; i++)
    scanf("%d", coin+i);
  solve(0, 0, 0);
  printf("%d\n", ans);
  return 0;
}
```

위 프로그램에서의 solve()함수는 다음과 같은 상태를 표현하고 있다.

solve(mon, k, cnt)= “k번째 이하의 동전을 cnt개 사용하여 mon원을 거슬러 준 상태”

이 방법은 앞에서 시도했던 방법보다 속도가 획기적으로 빨라진다. 그 이유는 전체 상태를 그려보면 앞의 방법보다 이번에 구조화한 방법의 정점의 수가 훨씬 적기 때문이다. 이와 같이 구조를 어떻게 설계하느냐에 따라 알고리즘의 계산량이 달라지기 때문에, 문제를 해결할 때 먼저 최적의 구조를 설계하는 것이 중요하다.

문제 17

예산 관리

정보 선생님은 예산이 많은 부서에서 일하고 있다. 학기말이 가까워지면서 부서의 예산을 가급적 모두 집행해야 될 상황이 되었다.

정보 선생님은 예산 범위를 넘지 않는 선에서 다양한 활동을 하고 싶어 한다.
지금 남은 예산(B)이 40이고(단위:만원), 예산을 사용할 수 있는 활동(n)이 6개가 있다.

6개의 활동에 각각 드는 비용은 7, 13, 17, 19, 29, 31이다. 여기서 40을 채울 수 있는 활동의 개수는 상관이 없다. 40을 넘지 않는 범위에서 활동 비용을 조합해보면,

7 + 13 + 17 = 37
7 + 31 = 38
7 + 13 + 19 = 39
...

따라서 40을 초과하지 않으면서 예산을 최대로 사용할 수 있는 비용은 39이다.
정보 선생님을 도와 줄 수 있는 프로그램을 작성하시오.

입력

첫째 줄에 남은 예산(B)이 입력된다. (10 <= B <= 35,000)
둘째 줄에 예산을 사용할 수 있는 활동의 수(n)가 입력된다. (1 <= n <= 21)
셋째 줄에 공백을 기준으로 n개의 활동비가 양의 정수로 입력된다.

출력

남은 예산을 초과하지 않으면서 최대로 사용할 수 있는 비용액을 출력한다.

입력 예	출력 예
40 6 7 13 17 19 29 31	39

풀이

이 문제는 전형적인 백트래킹에 의한 전체 탐색법을 연습할 수 있는 좋은 문제이다. 상황을 간단히 요약하면 예산 B를 넘지 않는 활동들의 최대 합을 구하는 것이다. 문제의 상황을 표로 정리하면 다음과 같다.

활동 번호	활동 비용	최대 합을 낼 수 있는 조합
1	7	●
2	13	●
3	17	
4	19	●
5	29	
6	31	

백트래킹 함수 f는 i번째 활동을 선택하느냐, 선택하지 않느냐의 두 갈래로 나누어진다. 이것은 곧 이진트리의 형태로 전개가 이루어지고, 계산량도 $O(2^n)$임을 의미한다.

백트래킹 함수 f의 정의를 명확하게 하는 것이 매우 중요한데, 다음과 같이 정의하였다.

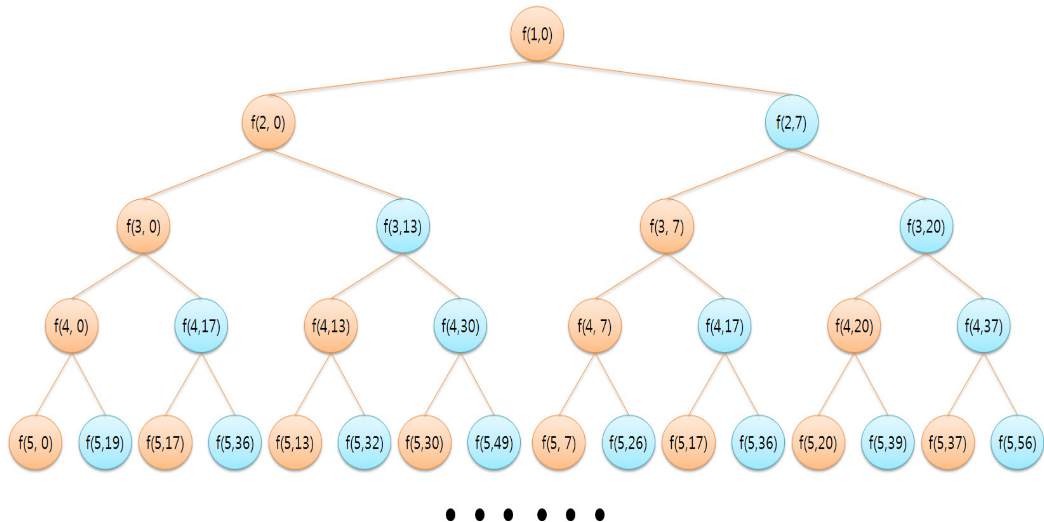
f(i, sum) = i번째 활동을 고려할 때, i-1까지 활동비용의 합계가 sum인 경우

전체 소스는 다음과 같다.

줄	코드	참고
1	`#include <stdio.h>`	5: i번 활동, i-1까지의 합계 13: i번째 활동을 포함하거나 14: 포함하지 않거나
2		
3	`int B, n, act[23], res;`	
4		
5	`void f(int i, int sum)`	
6	`{`	
7	`  if(i==n+1)`	
8	`  {`	
9	`    if(sum<=B && sum>res)`	
10	`      res=sum ;`	
11	`    return;`	

줄	코드	참고
12	`  }`	
13	`  f(i+1, sum+act[i]);`	
14	`  f(i+1, sum);`	
15	`}`	
16		
17	`int main()`	
18	`{`	
19	`  int i;`	
20	`  scanf("%d %d", &B, &n);`	
21	`  for(i=1; i<=n; i++)`	
22	`     scanf("%d", &act[i]);`	
23	`  f(1, 0);`	
24	`  printf("%d", res);`	
25	`  return 0;`	
26	`}`	

메인 함수에서 f(1, 0)을 호출하면 다음 구조로 백트래킹이 이루어진다.



9행은 이 값들 중 예산 B를 넘지 않으면서 최대 활동비용 sum의 값을 찾는 부분이다. 이 예제에서는 f(5, 39)에서 최댓값이 확인된다.

문제 18

0/1 배낭 문제(S)

어떤 배낭에 W무게만큼 물건을 담을 수 있다.
물건들은 (무게 wi, 가격 vi) 정보를 가지고 있는데, 물건들을 조합해서 담아 가격의 총합이 최대가 되게 하려고 한다.

물건들은 한 종류씩 밖에 없으며, 절대 배낭의 무게를 초과해서는 안 된다.

입력

첫째 줄에 물건의 개수 n(1<= n <= 100)과 배낭의 무게 w(1 <= w <= 10000)가 입력된다.
둘째 줄부터 n+1째줄 까지 물건들의 정보가 wi, vi가 한 줄에 하나씩 입력된다.
(1 <= wi, vi <= 100)

출력

배낭의 무게 W를 초과하지 않으면서 물건의 가격의 총합의 최댓값을 출력한다.

입력 예	출력 예
4 5 2 3 1 2 3 3 2 2	7

풀이

배낭 문제는 정보과학에 있어 유명한 문제 중 하나이다. 이 문제를 전체 탐색에서부터 접근해 보자. 위 문제의 상황을 표로 작성해보자.

물건 번호(i)	무게(wi)	가치(vi)
1	2	3
2	1	2
3	3	3
4	2	2

배낭의 무게가 5이므로, 5를 넘지 않으면서 채울 수 있는 방법 중 최고 가치를 가지는 경우를 전체 탐색하면 된다. 이 경우 1, 2, 4번의 물건을 담는 경우 W=5가 되고, 가치는 3 + 2 + 2 = 7로 최대가 된다. 우선 백트래킹에 의한 전체 탐색을 해보자.

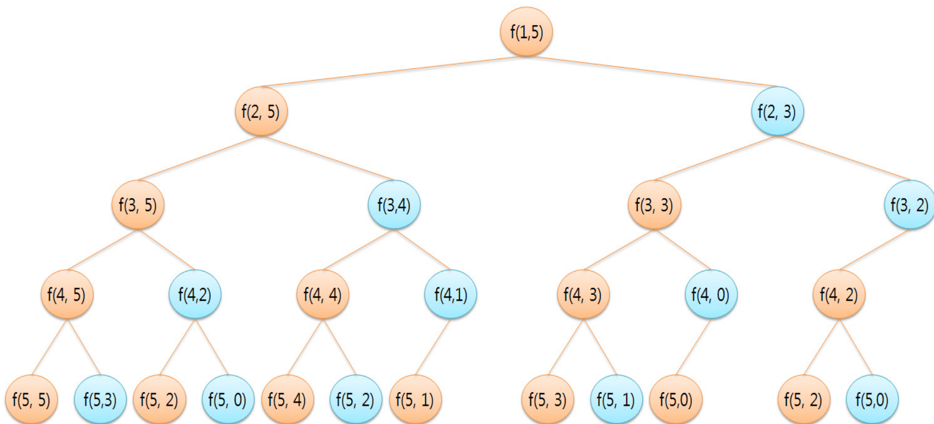
함수 f의 의미는 다음과 같다.

f(i, r) = i~n번째 물건까지 고려했을 때, 남은 무게가 r인 가치의 최대 합

수학적 식으로 정리하면 다음과 같다.

$$f(i,r)=\begin{cases}\max(f(i+1,r)\,\text{or}\,f(i+1,r-w[i])+v[i]) & (w[i]\le r)\\ f(i+1,r) & (w[i]>r)\end{cases}$$

이 함수를 이용하여 실제로 호출되는 과정은 다음과 같다.



이 트리에서 물건의 개수 n이 1증가할 때마다 트리의 깊이가 1씩 증가하므로, n이 커지면 계산량이 기하급수적으로 많아진다. 이 알고리즘은 i번째 물건을 넣느냐 넣지 않느냐로 나뉘기 때문에 전체 계산량은 $O(2^n)$이다.

줄	코드	참고
1	#include <stdio.h>	7: f(물건 번호 i, 남은 무게 r)
2		
3	int W, n, i, j, w[102], v[102];	
4		
5	int max(int a, int b) {return a>b ? a:b;}	
6		
7	int f(int i, int r)	
8	{	
9	if(i==n+1)	
10	return 0;	
11	else if(r<w[i])	
12	return f(i+1, r);	
13	else	
14	return max(f(i+1, r), f(i+1, r-w[i])+v[i]);	
15	}	
16		
17	int main()	
18	{	
19	scanf("%d %d", &n, &W);	
20	for(i=1; i<=n; i++)	
21	scanf("%d%d", &w[i], &v[i]);	
22	printf("%d", f(1, W));	
23	return 0;	
24	}	

22행의 메인함수에서 f(1, W)의 의미는 아직 1번 물건을 배낭에 넣지 않고, 무게가 W일 때 가치의 합을 구한다는 의미이다. 이렇게 호출된 함수 f는 i번째 물건을 배낭에 넣을지, 넣지 않을지를 결정하고 다음 i+1탐색으로 확장된다. 배낭의 무게보다 많이 넣으면 안 되므로 9~10행에서 배낭 무게를 넘지 못하도록 막고 있다.

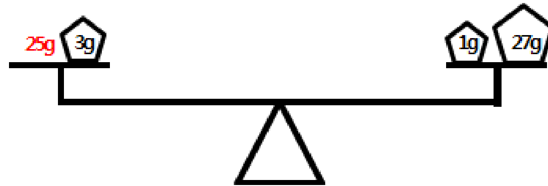
14행은 물건 가치의 최댓값을 구하는 부분으로 i번째 물건을 넣거나, 넣지 않았을 경우 중 최댓값으로 요약할 수 있고, 이 과정을 재귀적으로 i+1을 호출하여 이후 계산 결과를 합하여 최댓값을 구할 수 있다.

문제 19

저울 추(S)

평형저울을 이용하여 1kg 이하의 물건의 무게를 재려고 한다. 준비되어 있는 추는 1g, 3g, 9g, 27g, 81g, 243g, 729g과 같이 7개의 추뿐이다.

평형저울의 양쪽 접시에 물건과 추를 적절히 놓음으로써 물건의 무게를 잴 수 있는데, 예를 들어, 25g의 물건을 재기 위해서는 다음과 같이 저울에 올려놓으면 된다.



물건의 무게가 입력되었을 때 양쪽의 접시에 어떤 추들을 올려놓아야 평형을 이루는지를 결정하는 프로그램을 작성하시오.

입력

1. 물건의 무게를 나타내는 하나의 정수 n이 입력된다(1≤n≤1,000).
2. n은 물건의 무게가 몇 그램인지를 나타낸다.

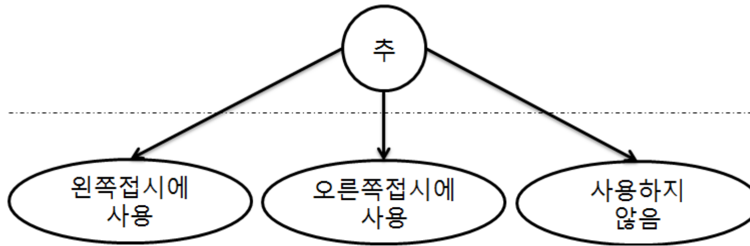
출력

1. 저울의 왼쪽 접시와 오른쪽 접시에 올린 추를 0으로 구분하여 출력한다.
2. 각 접시에 올린 추들을 무게가 가벼운 추부터 하나의 공백으로 구분하여 출력한다.
3. 물건의 무게를 왼쪽 접시의 처음에 표시한다.

입력 예	출력 예
25	25 3 0 1 27
40	40 0 1 3 9 27

풀이

이 문제는 전체탐색법으로 풀 수 있는 전형적인 문제로 볼 수 있다. 일단 추의 수가 8개이므로 매우 적다. 그리고 각 추의 사용법은 다음 중 하나이다.



즉 8개의 추에 대해서 3가지 사용방법에 대하여 깊이우선으로 탐색하면 가능 여부를 알 수 있다.

따라서 다음과 같은 소스코드로 깊이우선탐색을 할 수 있다.

코드	참고
`for(int i=0; i<7; i++)`	
`{`	
`   if(chk[i] == 0)`	
`   {`	
`      chk[i] = 1;`	
`      solve(n, sum+scale[i]);`	
`      chk[i] = 2;`	
`      solve(n+scale[i], sum);`	
`      chk[i] = 0;`	
`   }`	
`}`	

여기서 chk의 역할은 한 번 사용한 추를 다시 사용할 수 없기 때문에 한 번 사용한 추에 대한 사용 여부를 체크한다.

지금까지 체크를 할 때, 1은 “사용”, 0은 “사용하지 않음” 등으로 활용한 경우가 많은데 이 문제에서는 왼쪽, 오른쪽 접시 중 어느 접시에 올렸는지도 다루고 있다. 이와 같은 아이디어는 활용범위가 크므로 반드시 익혀둘 수 있도록 한다.

chk[k] = 0(k번째 추는 사용하지 않았음)
chk[k] = 1(k번째 추를 오른쪽 접시에 올렸음)
chk[k] = 2(k번째 추를 왼쪽 접시에 올렸음)

위의 아이디어로 구현한 소스코드는 다음과 같다.

코드	참고

```c
#include <stdio.h>
#include <stdlib.h>

int n, scale[8]={1,3,9,27,81,243,729}, chk[8], end;

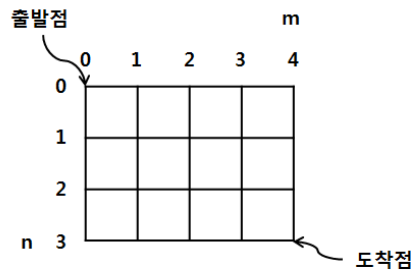
void solve(int n, int sum)
{
  if(end) return;
  if(sum==n)
  {
    for(int c=2; c>0; c--)
    {
      for(int i=0; i<7; i++)
        if(chk[i]==c)
          printf("%d ", scale[i]);
      if(c==2)
        printf("0 ");
    }
    end=1;
   }
   for(int i=0; i<7; i++)
   {
     if(chk[i]==0)
     {
       chk[i]=1, solve(n, sum+scale[i]);
       chk[i]=2, solve(n+scale[i], sum);
       chk[i] = 0;
     }
  }
}

int main()
{
  scanf("%d" ,&n);
  printf("%d ", n);
  solve(n, 0);
  return 0;
}
```

문제 20

격자길(S)

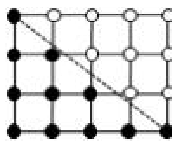
H*W 격자에서 왼쪽 위(0,0)에서 오른쪽 아래(H, W)까지 갈 수 있는 길의 수를 헤아리고자 한다.



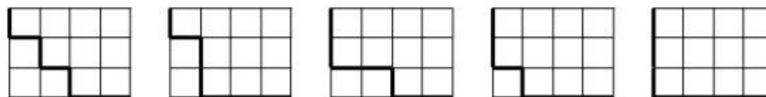
길을 갈 때 몇 가지 제약사항이 있다.

(1) 격자 위의 선을 따라간다.
(2) 아래쪽 또는 오른쪽으로만 갈 수 있다.
(3) (0,0)과 (H, W)를 잇는 대각선보다 위쪽에 있는 점들은 통과할 수 없다.
(대각선에 위치하는 점은 통과할 수 있다.)

아래의 그림에서 흰점은 통과할 수 없는 점이고 검은 점은 통과할 수 있는 점이다.



예를 들어, 3×4 격자에서 갈 수 있는 길은 다음과 같이 5가지가 있다.



격자의 크기가 입력되었을 때 (0,0)부터 (H, W)까지 갈 수 있는 길의 수를 출력하는 프로그램을 작성하시오.

격자길(S) (계속)

입력

1. 두 개의 정수 H와 W가 입력된다.
2. H는 격자의 세로 크기를, W는 격자의 가로 크기를 각각 나타낸다.

[입력값의 정의역]
1 <= H, W <= 10

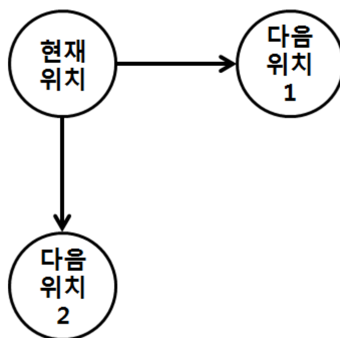
출력

(0,0)에서 (H, W)까지 갈 수 있는 길의 수를 출력한다.

입력 예	출력 예
3 4	5

풀이

이 문제는 전체탐색법으로 해결할 수 있다. 일단 현재 지점에서 이동 가능한 방법은 다음과 같은 2가지이다.



단, [다음위치1]로 갈 때는 항상 가능한 것은 아니기 때문에 갈 수 있는지 판단하는 기준을 정해야 한다. 만약 폭이 W이고 높이가 H라면 이동하려고 하는 정점의 좌표가 (w, h)라고 하면 다음 조건을 만족해야지만 이동가능한 정점이다.

$$\frac{H}{W} \le \frac{h}{(w+1)} \qquad (\text{단, } H,\ W,\ h,\ w\text{는 모두 실수})$$

위 조건을 만족하면 [다음위치1]로 이동가능하고 아니면 위로 이동하는 경로가 없다. 따라서 다음과 같은 탐색함수를 구현하는 코드를 작성할 수 있다.

코드	참고
<pre>void solve(int h, int w) { solve(h+1, w); if((double)H/W<=(double)h/(w+1)) solve(h, w+1); }</pre>	

위 함수를 기반으로 전체탐색법으로 작성한 소스코드는 다음과 같다.

줄	코드	참고

```c
#include <stdio.h>

int H, W, ans;

void solve(int h, int w)
{
  if(h>H || w>W) return;
  if(h==H && w==W)
  {
    ans++;
    return;
  }
  solve(h+1, w);
  if((double)H/W<=(double)h/(w+1))
    solve(h, w+1);
}

int main()
{
  scanf("%d%d",&H,&W);
  solve(0, 0);
  printf("%d\n", ans);
  return 0;
}
```

문제 21

선물(S)

길동이는 세쌍둥이의 첫째이다. 길순이가 둘째이고, 길삼이가 막내이다. 길동 3남매의 생일을 맞이하여 전국 각지에서 친지들이 보내온 수많은 선물이 도착하였다.

길동이 부모는 이 선물들을 길동이 3남매에게 어떻게 나누어 줄 것인가로 고민하고 있다. 선물의 크고 작음 때문에 발생될 수도 있는 남매간의 다툼을 미연에 방지하고자 길동이 가족은 다음과 같이 나누기로 결정하였다.

(1) 선물의 내용을 미리 보지 않고 부피만을 기준으로 배분한다.
(2) 한 사람이 가지는 선물의 개수는 배분의 기준이 아니다.
(3) 선물이 공평하게 나누어질 수 있도록 3남매가 가지는 선물들의 부피의 합계 차이가 최소가 되도록 한다.
(4) 선물의 부피가 똑같이 나누어지지 못하는 경우에는 길동-길순-길삼의 순으로 합계 부피가 많도록 배분한다.
(5) 3남매가 가지게 되는 부피가 결정되면, 길삼-길순-길동의 순으로 선물을 선택한다.

우리가 길동 부모의 수고를 덜어주고자 길동이 3남매가 가지게 될 선물의 부피를 계산하고자 한다. 선물 부피에 따른 선물 배분의 세부적인 조건은 다음과 같다.

조건 1: 아래의 d가 최소가 되도록 한다.

d = (길동 선물의 부피 합) - (길삼 선물의 부피 합)

조건 2: 같은 d가 되는 배분 방법이 여럿 존재하는 경우에는 길동의 선물의 부피 합이 적은 방법을 선택한다.

선물(S) (계속)

예를 들어, 선물이 6개이고 그 부피가 다음과 같다면,

6, 4, 4, 4, 6, 9

길동은 부피의 합계가 12, 길순은 12, 길삼은 9를 가지도록 배분하면 조건 1에 따라 12-9=3로 최소가 된다.

(길동 13, 길순 10, 길삼 10으로 배분하는 방법도 13-10=3으로 차이가 3이 되지만, 조건 2에 따라 답이 되지 못한다.)

선물의 부피가 입력되었을 때 3남매에게 나누어줄 선물의 합계 부피를 구하는 프로그램을 작성하시오.

입력

1. 첫 줄에 선물의 개수를 나타내는 정수 n가 입력된다(3≤n≤15).
2. 다음 줄에 선물의 부피를 나타내는 n개의 정수가 공백으로 분리되어 입력된다.
3. 선물의 부피는 0보다 크고 100보다 작다

출력

1. 길동 3남매가 가지게 될 선물의 합계 부피를 출력한다.
2. 길동, 길순, 길삼의 순으로 3개의 정수를 하나의 공백으로 분리하여 출력한다.

입력 예	출력 예
6 6 4 4 4 6 9	12 12 9
3 2 10 1	10 2 1
9 1 1 1 4 6 1 1 1 1	6 6 5

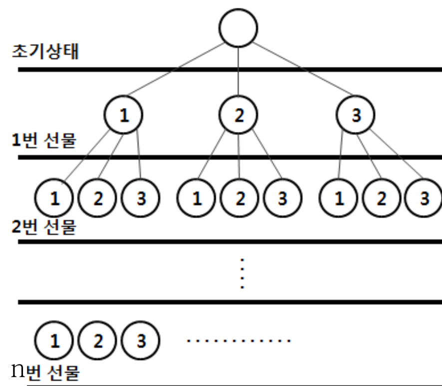
풀이

일단 주어진 최대 선물 수가 15인 것을 감안한다면, 전체탐색법으로 충분히 해결할 수 있는 수이다.

최대 선물의 수가 15개이고 이를 3명 중 한 명이 반드시 가져야 하므로 나올 수 있는 모든 경우의 수는

$3^{15} = 14,348,907$ 개로 해결 가능한 범위라고 할 수 있다.

먼저 백트래킹을 구성할 탐색공간트리를 나타내면 다음과 같다.



위의 트리의 경로를 따라서 n번 선물까지 지불하여, 1의 합, 2의 합, 3의 합을 각각 구하면 길동, 길순, 길삼이 받는 선물의 부피가 된다. 이 중 부피의 차가 가장 적은 것을 출력하면 해가 된다.

이 과정을 함수로 표현한 코드는 다음과 같다.

코드	참고
`void solve( int no, int a, int b, int c )`	

```
void solve( int no, int a, int b, int c )
{
  if(no<n)
  {
    solve(no+1, a, b, c+gift[no]);
    solve(no+1, a, b+gift[no], c);
    solve(no+1, a+gift[no], b, c);
  }
}
```

주어진 함수를 기반으로 전체탐색법으로 작성한 소스코드는 다음과 같다.

```
#include <stdio.h>
#include <algorithm>

int gift[30], chk[30], A, B, C, n, S;

int comp(int a, int b)
{
   return a>b;
}

void solve(int no, int a, int b, int c)
{
  if(no<n)
  {
    solve(no+1, a, b, c+gift[no]);
    solve(no+1, a, b+gift[no], c);
    solve(no+1, a+gift[no], b, c);
  }
  else if(a>=b && b>=c)
  {
    A=a, B=b, C=c;
  }
}

int main()
{
  int i;
  scanf("%d", &n);
  for(i=0; i<n; S+=gift[i++])
    scanf("%d", &gift[i]);
  std::sort(gift, gift+n, comp);
  solve(0, 0, 0, 0);
  printf("%d %d %d\n", A, B, C);
  return 0;
}
```

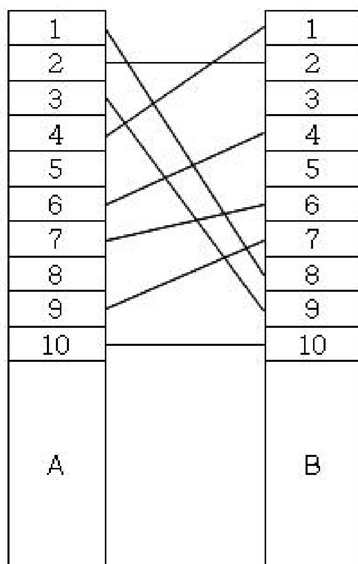
하지만 n이 20정도가 되면 이 코드로는 해를 구할 수 없다.

문제 22

전깃줄(S)

두 전봇대 A와 B 사이에 하나 둘씩 전깃줄을 추가하다 보니 전깃줄이 서로 교차하는 경우가 발생하였다. 합선의 위험이 있어 이들 중 몇 개의 전깃줄을 없애 전깃줄이 교차하지 않도록 만들려고 한다.

예를 들어, <그림 1>과 같이 전깃줄이 연결되어 있는 경우 A의 1번 위치와 B의 8번 위치를 잇는 전깃줄, A의 3번 위치와 B의 9번 위치를 잇는 전깃줄, A의 4번 위치와 B의 1번 위치를 잇는 전깃줄을 없애면 남아있는 모든 전깃줄이 서로 교차하지 않게 된다.



< 그림 1 >

전깃줄이 전봇대에 연결되는 위치는 전봇대 위에서부터 차례대로 번호가 매겨진다. 전깃줄의 개수와 전깃줄들이 두 전봇대에 연결되는 위치의 번호가 주어질 때, 남아있는 모든 전깃줄이 서로 교차하지 않게 하기 위해 없애야 하는 전깃줄의 최소 개수를 구하는 프로그램을 작성하시오.

전깃줄(S) (계속)

입력

첫째 줄에는 두 전봇대 사이의 전깃줄의 개수가 주어진다. 전깃줄의 개수는 100이하의 자연수이다.

둘째 줄부터 한 줄에 하나씩 전깃줄이 A전봇대와 연결되는 위치의 번호와 B전봇대와 연결되는 위치의 번호가 차례로 주어진다.

위치의 번호는 500 이하의 자연수이고, 같은 위치에 두 개 이상의 전깃줄이 연결될 수 없다.

출력

첫째 줄에 남아있는 모든 전깃줄이 서로 교차하지 않게 하기 위해 없애야 하는 전깃줄의 최소 개수를 출력한다.

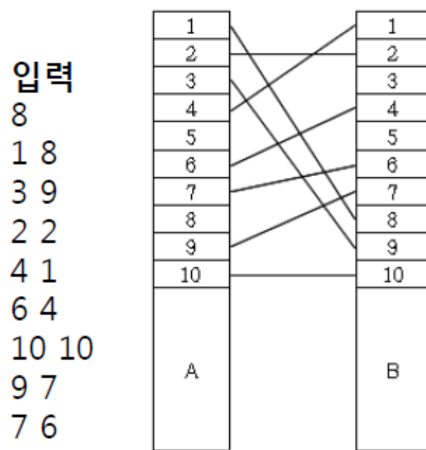
입력 예	출력 예
8 1 8 3 9 2 2 4 1 6 4 10 10 9 7 7 6	 3

출처: 한국정보올림피아드(2007 지역본선 초등부)

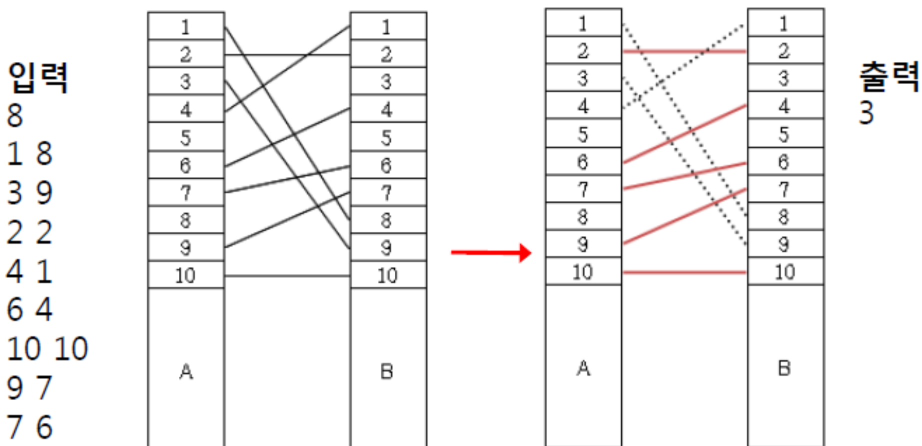
풀이

두 전봇대 A, B에 연결된 전깃줄이 서로 교차하지 않도록 전깃줄을 제거하는 문제이다. 단, 최소의 전깃줄을 제거해야 한다. 전깃줄의 개수는 100,000 이하이며 위치번호는 500,000 이하 자연수이다. 또한, 같은 위치에 전깃줄은 없다.

입력이 왼쪽과 같다면 다음 그림은 상태를 표현한 것이다.



A 전봇대에서 1, 3, 4 번 전깃줄을 제거하면 나머지 모든 전깃줄이 서로 교차하지 않게 되고, 최소 제거 개수는 3이 된다.



만약, “가장 많이 교차하는 전깃줄을 제거해 가면서 교차여부를 판단해보면? 쉽게 해결할 수 있을 것이다”라는 가정을 한다면,

A 기둥에 연결되어있는 각 전깃줄에 교차횟수를 계산하고 기록한 후, 가장 많은 교차 횟수를 가진 것부터 제거해 가면서 교차 여부를 확인하는 방법을 사용할 수 있다.

교차수	A	B
5	1	1
2	2	2
4	3	3
3	4	4
0	5	5
2	6	6
2	7	7
0	8	8
2	9	9
0	10	10

#1

교차수 최대인 전깃줄 1 제거

교차수	A	B
0	1	1
1	2	2
4	3	3
2	4	4
0	5	5
1	6	6
1	7	7
0	8	8
1	9	9
0	10	10

제거 후 교차수 재계산

교차수	A	B
0	1	1
1	2	2
4	3	3
2	4	4
0	5	5
1	6	6
1	7	7
0	8	8
1	9	9
0	10	10

#2

교차수 최대인 전깃줄 4 제거

교차수	A	B
0	1	1
1	2	2
0	3	3
1	4	4
0	5	5
0	6	6
0	7	7
0	8	8
0	9	9
0	10	10

제거 후 교차수 재계산

교차수	A	B
0	1	1
0	2	2
0	3	3
0	4	4
0	5	5
0	6	6
0	7	7
1	8	8
0	9	9
1	10	10

교차수 최대인 전깃줄 8 제거

교차수	A	B
0	1	1
0	2	2
0	3	3
0	4	4
0	5	5
0	6	6
0	7	7
0	8	8
0	9	9
0	10	10

제거 후 교차수 재계산

완성????

3개를 제거하면 2개가 남지만, 2개를 제거하고 3개를 남기는 방법도 있다. 따라서 교차수가 가장 많은 전깃줄을 순서대로 제거해 나가는 방식은 적당하지 않다.

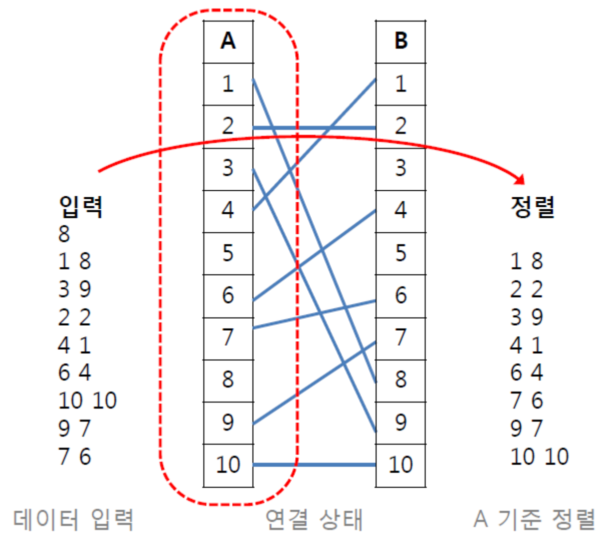
교차수	A	B
	1	1
	2	2
	3	3
	4	4
	5	5
	6	6
	7	7
	8	8
	9	9
	10	10

3개 제거 방법?

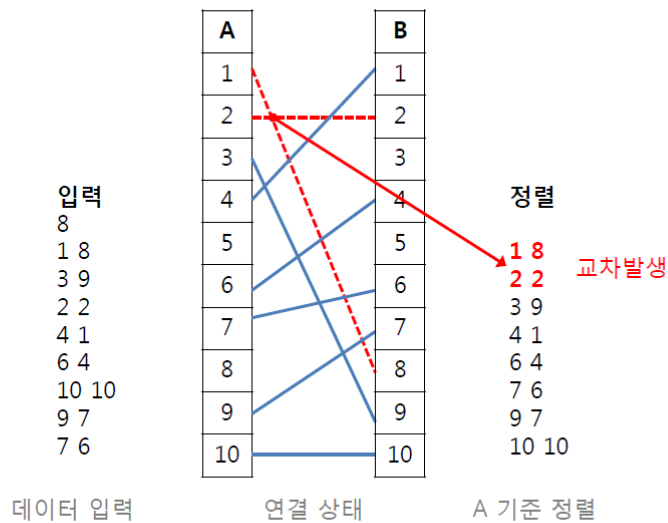
교차수	A	B
	1	1
	2	2
	3	3
	4	4
	5	5
	6	6
	7	7
	8	8
	9	9
	10	10

하지만! 2개만 제거해도 된다!!

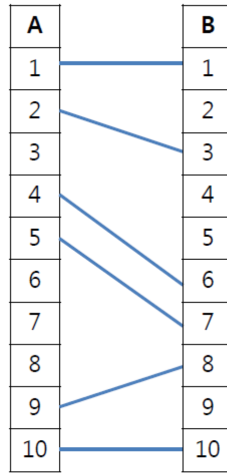
두 전봇대의 전기줄의 연결 순서를 생각하고, 한 전봇대를 기준으로 연결 관계를 생각해보면, 보다 간단하고 정확한 문제해결 전략을 만들어낼 수 있다.



A를 기준으로 두 쌍의 데이터를 정렬하고 관찰해 보면, B 에서 위에서 아래로 오름차순이 아닌 경우 두 전깃줄이 서로 교차됨을 확인할 수 있다.



따라서 만약, A에 연결되어 있는 순서가 오름차순이고, B에 연결되어 있는 순서도 모두 오름차순이라면? 두 기둥에 연결되어있는 전깃줄은 서로 교차하지 않게 된다는 것을 의미한다.



상태

1 1
2 3
4 6
5 7
9 8
10 10

교차 없음! 내림차순으로 구성됨.

연결 상태　　　A 기준 정렬

따라서 입력된 데이터를 A를 기준으로 정렬시킨 후, B의 데이터로 만들 수 있는 가장 오름 수열을 찾아내면, 그 수열의 길이가 바로 서로 교차하지 않는 전깃줄의 최대 개수가 되는 것이다.

입력
8
1 8
3 9
2 2
4 1
6 4
10 10
9 7
7 6

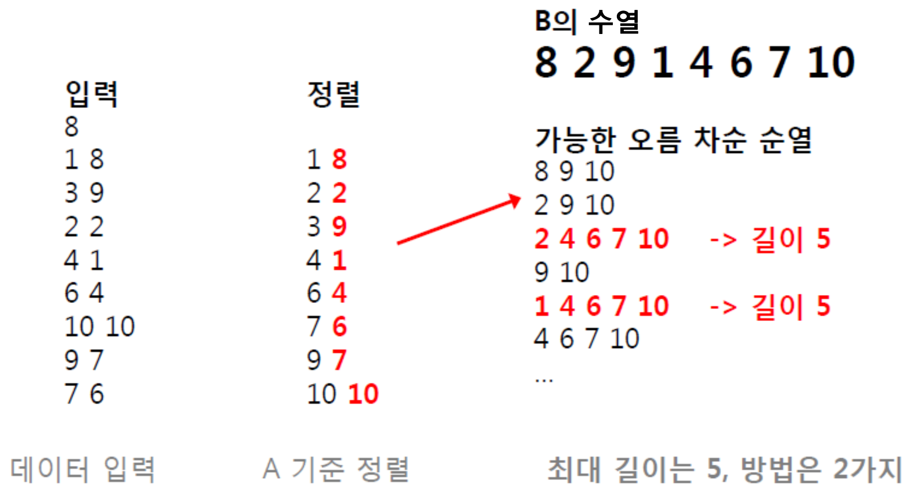
정렬
1 8
2 2
3 9
4 1
6 4
7 6
9 7
10 10

B의 수열
8 2 9 1 4 6 7 10

가능한 오름 차순 순열
8 9 10
2 9 10
2 4 6 7 10 -> 길이 5
9 10
1 4 6 7 10 -> 길이 5
4 6 7 10
...

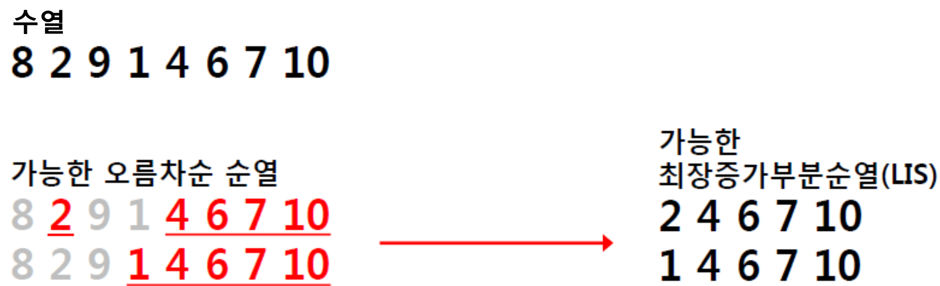
데이터 입력　　　A 기준 정렬

따라서 A를 기준으로 정렬한 후, B의 수열로 가능한 모든 오름차순 수열 중에서 가장 긴 길이의 수열을 찾아내면, 그 길이는 가능한 최대를 의미하고, 가능한 수열 각각은 서로 다른 한 가지씩의 방법을 의미하게 된다.



이와 같이, 주어진 어떤 수열에서 순서대로 증가하면서 커지는 가장 긴 수열을 최장증가 부분수열(longest increasing subsequence, 이하 LIS)이라고 한다.

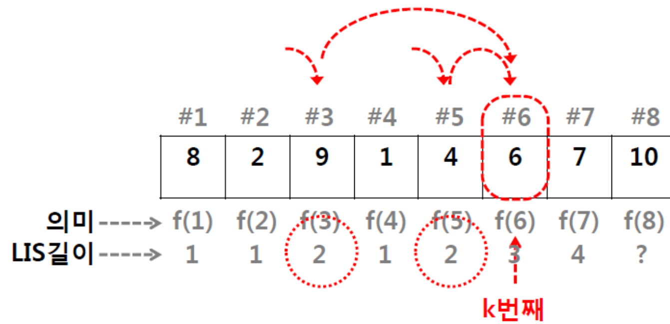
LIS의 길이를 입력된 데이터 쌍의 개수에서 빼면, 최대로 가능한 교차하지 않는 연결 개수를 구할 수 있게 된다.



LIS를 찾아내는 다양한 방법을 생각하고 만들어낼 수 있지만 재귀적 문제해결사고를 활용할 수도 있다.

1. k번째 위치를 마지막으로 만들어질 수 있는 LIS의 길이를 solve(k)라고 하자.

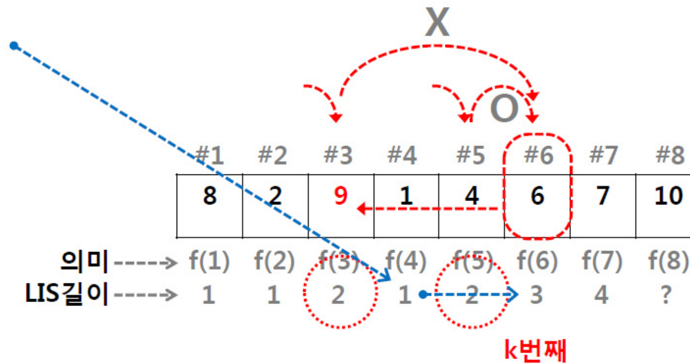
2. 그렇다면 solve(k)는 그 이전의 solve(k−1), solve(k−2), solve(k−3), … , solve(3), solve(2), solve(1)들의 길이에 1을 더한 것이라고 생각할 수 있다.



k번째가 마지막으로 구성되는 LIS라고 한다면?

3. 단, 오름차순으로 증가해야 하기 때문에 "k위치 값" 〉 "이전 위치 값" 의 조건을 만족해야 하고,

4. 주의 깊게 생각해야 하는 것은 어떤 "이전위치까지의 길이"에 1을 더했을 때, 오히려 더 작은 경우가 생길 수 있다는 것이다.



k번째가 마지막으로 구성되는 LIS라고 한다면?

각 k번째 위치를 마지막으로 하는 LIS길이를 모두 찾은 후, 그 중에서 가장 큰 값(LIS의 길이)을 입력된 개수에서 빼면 된다.



모든 k번째 위치를 마지막으로 하는 LIS길이 중에서 최댓값을 이용하면 된다.

```
#include <stdio.h>
#include <algorithm>

struct w {int a; int b;} d[100001];
int n;
bool cmp(w x, w y) {return x.a<y.a;}
int max(int p, int q) {return p>q ? p:q;}

int solve(int k)
{
  int i,count=1;
  for(i=k-1; i>=1; i--)
    if(d[k].b > d[i].b) count=max(count,solve(i)+1);
  return count;
}

int main()
{
  int i, t, m=0;
  scanf("%d", &n);
  for(i=1; i<=n; i++)
    scanf("%d %d", &d[i].a, &d[i].b);
  std::sort(d+1, d+n+1, cmp);
  for(i=1; i<=n; i++)
  {
    t=solve(i);
    if(m<t) m=t;
  }
  printf("%d\n", n-m);
  return 0;
}
```

LIS를 찾아내는 방향을 바꿔서 생각해 볼 수도 있다.

1. k번째 위치를 첫 번째로 시작해서 만들 수 있는 LIS의 길이를 $f(k)$라고 하자.

2. 그렇게 생각하면 1~n개가 있을 때의 LIS의 길이는 $f(1)$로 생각할 수 있다.
그렇게 $f(1)$, $f(2)$, $f(3)$, … , $f(n-2)$, $f(n-1)$, $f(n)$을 모두 구하면, 그 중 가장 큰

값이 최대로 가능한 LIS의 길이가 된다.

	#1	#2	#3	#4	#5	#6	#7	#8
	8	2	9	1	4	6	7	10
의미 ----->	f(1)	f(2)	f(3)	f(4)	f(5)	f(6)	f(7)	f(8)
LIS길이 ----->	3	5	2	5	4	3	2	1

k번째

k번째를 첫 번째로 시작하는 LIS라고 한다면?
k번째가 마지막으로 구성되는 LIS라고 한다면?

첫 번째 풀이에서 LIS 길이를 찾는 방향을 반대로 한 코드의 형태는 다음과 같다.

```
#include <stdio.h>
#include <algorithm>

struct w{int a; int b;} d[100001];
int n;
bool cmp(w x, w y){return x.a < y.a;}
int max(int p, int q){return p>q ? p:q;}

int solve(int k)
{
  int i,count=1;
  for(i=k+1; i<=n; i++)
    if(d[k].b<d[i].b) count=max(count,solve(i)+1);
  return count;
}

int main()
{
  int i, t, m = 0;
  scanf("%d", &n);
  for(i=1; i<=n; i++)
    scanf("%d %d", &d[i].a, &d[i].b);
  std::sort(d+1, d+n+1, cmp);
  for(i=1; i<=n; i++)
  {
    t=solve(i);
    if(m<t) m = t;
  }
  printf("%d\n", n-m);
  return 0;
}
```

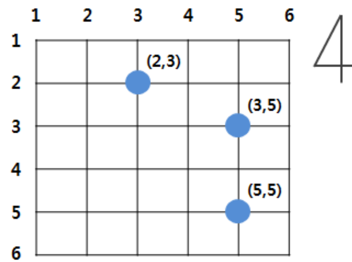
문제 23

경찰차(S)

어떤 도시의 중심가는 n개의 동서방향 도로와 n개의 남북방향 도로로 구성되어 있다.

모든 도로에는 도로 번호가 있으며 남북방향 도로는 왼쪽부터 1에서 시작하여 n까지 번호가 할당되어 있고 동서방향 도로는 위부터 1에서 시작하여 n까지 번호가 할당되어 있다. 또한 동서방향 도로 사이의 거리와 남북방향 도로 사이의 거리는 모두 1이다.

동서방향 도로와 남북방향 도로가 교차하는 교차로의 위치는 두 도로의 번호의 쌍인 (동서방향 도로 번호, 남북방향 도로 번호)로 나타낸다. n이 6인 경우의 예를 들면 다음과 같다.



이 도시에는 두 대의 경찰차가 있으며 두 차를 경찰차1과 경찰차2로 부른다. 처음에는 항상 경찰차1은 (1, 1)의 위치에 있고 경찰차2는 (n, n)의 위치에 있다.

경찰 본부에서는 처리할 사건이 있으면 그 사건이 발생된 위치를 두 대의 경찰차 중 하나에 알려 주고, 연락 받은 경찰차는 그 위치로 가장 빠른 길을 통해 이동하여 사건을 처리한다(하나의 사건은 한 대의 경찰차가 처리한다.).

그리고 사건을 처리 한 경찰차는 경찰 본부로부터 다음 연락이 올 때까지 처리한 사건이 발생한 위치에서 기다린다. 경찰 본부에서는 사건이 발생한 순서대로 두 대의 경찰차에 맡기려고 한다.

처리해야 될 사건들은 항상 교차로에서 발생하며 경찰 본부에서는 이러한 사건들을 나누어 두 대의 경찰차에 맡기되, 두 대의 경찰차들이 이동하는 거리의 합을 최소화 하도록 사건을 맡기려고 한다.

경찰차(S) (계속)

예를 들어 앞의 그림처럼 n=6인 경우, 처리해야 하는 사건들이 3개 있고 그 사건들이 발생된 위치를 순서대로 (3, 5), (5, 5), (2, 3)이라고 하자.

(3, 5)의 사건을 경찰차2에 맡기고 (5, 5)의 사건도 경찰차2에 맡기며, (2, 3)의 사건을 경찰차1에 맡기면 두 차가 이동한 거리의 합은 4 + 2 + 3 = 9가 되고, 더 이상 줄일 수는 없다.

처리해야 할 사건들이 순서대로 주어질 때, 두 대의 경찰차가 이동하는 거리의 합을 최소화 하는 프로그램을 작성하시오.

입력

입력 파일의 첫째 줄에는 동서방향 도로의 개수를 나타내는 정수 n(3≤n≤1,000)이 주어진다.

둘째 줄에는 처리해야 하는 사건의 개수를 나타내는 정수 w(1≤w≤15)가 주어진다.

셋째 줄부터 (w+2)번째 줄까지 사건이 발생된 위치가 한 줄에 하나씩 주어진다. 경찰차들은 이 사건들을 주어진 순서대로 처리해야 한다.

각 위치는 동서방향 도로 번호를 나타내는 정수와 남북방향 도로 번호를 나타내는 정수로 주어지며 두 정수 사이에는 빈 칸이 하나 있다. 두 사건이 발생한 위치가 같을 수 있다.

출력

첫째 줄에 두 경찰차가 이동한 총 거리를 출력한다.

입력 예	출력 예
6 3 3 5 5 5 2 3	9

출처: 한국정보올림피아드(2003 전국본선 중등부)

풀이

이 문제는 지금까지의 문제들과는 달리 상태를 정의하기가 쉽지 않기 때문에 그 만큼 난이도가 높은 문제이다.

하지만 이 문제도 입력값 n이 15이기 때문에 전체탐색법을 해결할 수 있다. 이 문제에서는 탐색을 구조화하는 방법이 중요한데, 사건을 순서대로 처리해야 하기 때문에 사건을 탐색의 기준으로 두고 처리할 수 있다.

각 사건을 처리하기 전 각 경찰차의 위치를 이용하여 상태를 지정할 수 있다. 실제 경찰차가 위치할 수 있는 위치는 $1{,}000 \times 1{,}000$개가 가능하지만, 사실 경찰차는 사건이 일어난 위치에만 있다고 가정해도 문제를 해결하는 데 지장이 없으므로, 실제로 경찰차가 위치할 수 있는 가능성은

$$(w\text{개} + 1\text{번 경찰차의 초기 위치} + 2\text{번 경찰차의 초기 위치})$$

로 모두 $w+2$곳뿐이다.

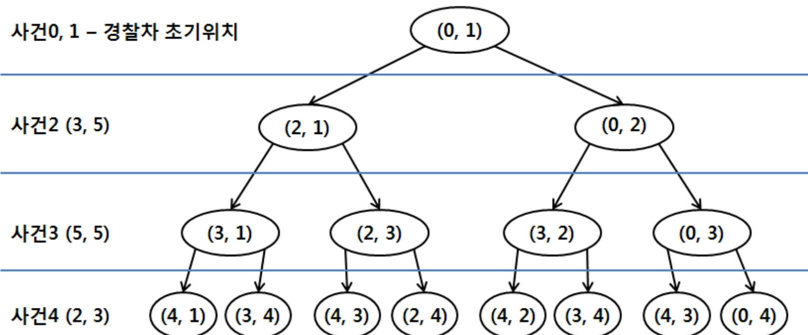
따라서 이 문제는 실제 사건의 수를 $w+2$ 개로 설정하고 시작한다. 2개의 사건은 처음 경찰차 2대가 있는 위치도 이미 사건이 발생했고, 이를 각 경찰차가 처리한 상태로 두기 위함이다. 이 문제를 해결하기 위하여 정점의 상태를 다음과 같이 정의할 수 있다.

solve(a, b, d) =
“max(a, b)번 사건까지 처리하면서 d만큼 이동한 후, 1번 경찰차는 a사건의 위치에, 2번 경찰차는 b사건의 위치에 있는 상태”

그리고 이 상태에 적용될 부등식 및 사건의 진행 순서를 결정하기 위해서 필요한 정보는 다음과 같다.

$$0 \le a,\ b < w+2,\ \text{다음사건} = \max(a,\ b)+1$$

만약 주어진 발생한 사건이 3건이고 각 사건의 순서별 위치는 (3, 5), (5, 5), (2, 3)이라면 전체탐색은 다음과 같이 진행된다.



위 구조에서 마지막 상태들 중 거리의 합이 가장 적은 것을 선택한다.

위의 상태를 깊이우선탐색을 이용하여 만든 소스코드는 다음과 같다.

줄	코드	참고

```
#include <stdio.h>

int E[1010][2], n, m, ans=987654321;

int min(int a, int b)
{
  return a>b? b:a;
}
int abs(int a)
{
  return a>0? a:-a;
}
int dis(int a, int b)
{
  return abs(E[a][0]-E[b][0])+abs(E[a][1]-E[b][1]);
}
int main()
{
  scanf("%d%d", &n, &m);
  E[0][0]=E[0][1]=1;
  E[1][0]=E[1][1]=n;
  for(int i=2; i<m+2; i++ )
    scanf("%d%d",&E[i][0], &E[i][1]);
  solve(0, 1, 0);
  printf("%d", ans);
  return 0;
}
```

배열 E는 각 사건의 위치를 저장한다. 20, 21행은 처음 경찰차의 초기위치를 사건 0, 1로 설정하는 코드이다.

따라서 사건의 수는 m+2개가 된다. 다음으로 min은 두 값 중 작은 값을, abs는 절댓값을 구하는 함수로 각 사건들 간의 거리를 구하는 데 이용된다.

줄	코드	참고
1	`void solve(int a, int b, int d)`	
2	`{`	
3	`  int next=(a>b? a:b)+1;`	
4	`  if(next>=m+2)`	
5	`  {`	
6	`    if(d<ans) ans=d;`	
7	`    return;`	
8	`  }`	
9	`  solve(next, b, d+dis(a, next));`	
10	`  solve(a, next, d+dis(b, next));`	
11	`}`	

next는 다음에 일어날 사건을 구하기 위한 변수로 a, b중 더 큰 값이 바로 이전 사건이므로 max(a, b) + 1이 다음 사건이 되며, 2행과 같은 방법으로 구한다.

위 알고리즘에서 두 사건간의 거리를 구할 때, 탐색할 때마다 사건 간의 거리를 계산하는데 이미 했던 계산을 계속 반복하는 경우가 발생한다. 이를 방지하기 위해서 미리 처음에 모든 사건들 간의 거리를 계산해 두고 이를 바로 불러 쓰는 방법이 있다. 소스코드는 다음과 같다.

줄	코드	참고
1	`#include<stdio.h>`	
2		
3	`int dis[1002][1002];`	
4		
5	`int main(void)`	
6	`{`	
7	`            :`	
8	`  for(i=0; i<m+2; i++)`	
9	`    for(j=0; j<m+2; j++)`	
10	`      dist[i][j]=abs(a[i][0]-a[j][0])+abs(a[i][1]-a[j][1]);`	
11	`            :`	
12	`}`	

이렇게 중복되는 계산을 줄이기 위하여 미리 표에 계산해 두는 방법을 이용하여 문제 전체를 해결하는 방법도 있다. 이 방법은 뒤에서 자세히 다루므로 여기서는 생략한다. 이와 같이 작성했을 경우에는 solve함수는 다음과 같이 달라진다.

```
void solve(int a, int b, int d)
{
  int next=(a>b ? a:b)+1;
  if(next>=m+2)
  {
    if(d<ans) ans=d;
    return;
  }
  solve(next, b, d+dis[a][next]);
  solve(a, next, d+dis[b][next]);
}
```

중복되는 계산이 많을수록 이 방법을 이용할 경우 효율이 좋아진다. 하지만 이 문제에서는 크게 이득은 아니지만 앞으로 이러한 방법을 활용할 수 있는 문제들이 많이 등장하므로 한 번쯤 익혀둘 가치는 있다.

문제 24

좋은 수열

숫자 1, 2, 3으로만 이루어지는 수열이 있다. 임의 길이의 인접한 두 개의 부분 수열이 동일한 것이 있으면, 그 수열을 나쁜 수열이라고 부른다. 그렇지 않은 수열은 좋은 수열이다.

다음은 나쁜 수열의 예이다.

33
32121323
123123213

다음은 좋은 수열의 예이다.

2
32
32123
1232123

길이가 n인 좋은 수열들을 n자리의 정수로 보아 그 중 가장 작은 수를 나타내는 수열을 구하는 프로그램을 작성하라. 예를 들면, 1213121과 2123212는 모두 좋은 수열이지만 그 중에서 작은 수를 나타내는 수열은 1213121이다.

입력

입력파일은 숫자 n 하나로 이루어진다. n은 1 이상 80 이하이다.

출력

화면에 1, 2, 3으로만 이루어져 있는 길이가 n인 좋은 수열들 중에서 가장 작은 수를 나타내는 수열만을 출력한다. 수열을 이루는 1, 2, 3들 사이에는 빈 칸을 두지 않는다.

입력 예	출력 예
7	1213121

출처: 한국정보올림피아드(1997 전국본선 중등부)

풀이

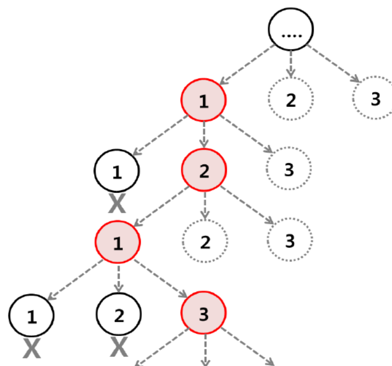
문제해결을 위한 핵심적인 알고리즘은 다음과 같이 구성해 볼 수 있다. 수열을 만들 때 1부터 시작해서 1, 2, 3 숫자를 하나씩 추가한다. 오름차순으로 수열을 만들고 좋은 수열을 판단하는 방법은 다음과 같은 과정으로 진행한다.

1. 새로운 수열을 만든다.
2. 만들어진 수열이 좋은 수열인지 평가한다.
3. 만든 수열이 n 자리인지 확인한다.
4. 아니라면 1,2,3을 순서대로 붙여보고 다시 확인한다.

위의 과정으로 수열을 만들어 가면 오름차순으로 가장 빠른 순서로 만들 수 있는 좋은 수열을 찾을 수 있다. 예를 들어 길이가 4인 좋은 수열을 위 방법대로 구하는 순서는 다음과 같다.

1 → 좋은 수열, 하지만 자릿수가 1
11 → 나쁜 수열
12 → 좋은 수열, 자릿수 1
121 → 좋은 수열, 자릿수 3
1211 → 나쁜 수열
1212 → 나쁜 수열
1213 → 좋은 수열, 자릿수 4 → 찾았다!

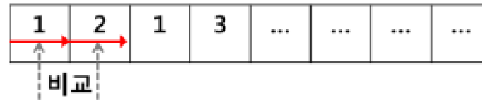
이 방법을 상태로 정의하고 구조를 만들면 다음과 같이 그려지게 된다. 최대 깊이가 80인 트리를 그려내야 하는데, 선택할 수 있는 곳들 중에서 X 부분은 나쁜 수열이기 때문에 그만 뻗어나가는 경우로 일단 한 번 나쁜 수열이라면, 이후에 어떤 것을 붙이든지 나쁜 수열이 되기 때문이다.

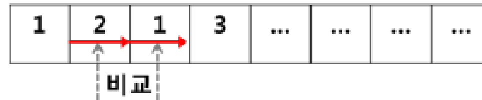


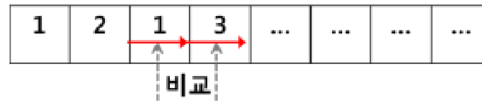
이 과정으로 진행해가면서 길이가 n이 되는 처음 값이 문제에서 찾는 해가 된다. 따라서 한 단계를 진행할 때마다 현재까지 만들어진 수열이 좋은 수열이 아닌지 판단하기 위한 방법을 만들면 된다.

어떤 수열이 좋은 수열인지 판단하기 위한 가장 간단한 방법은 앞에서부터 차례로, 1개가 연속된 경우, 2개가 연속된 경우, 3개가 연속된 경우, ... , $\frac{n}{2}$개가 연속된 경우를 모두 확인해 보는 방법이다.

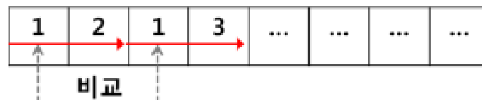
수열의 길이가 n이라면 비교하는 크기가 $\frac{n}{2}$일 때까지 해 보면 된다.

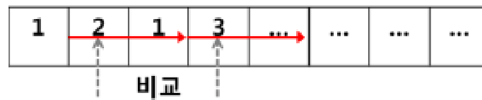


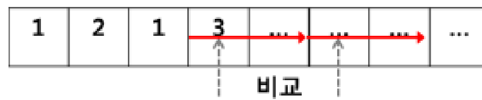




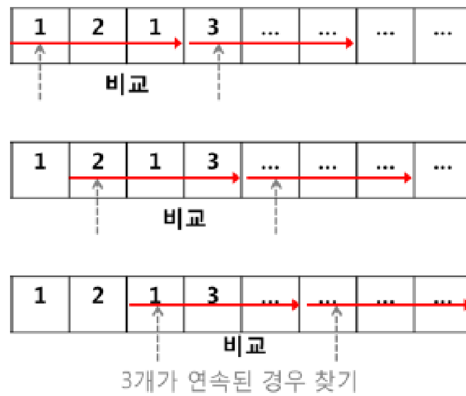
1개가 연속된 경우 찾기







2개가 연속된 경우 찾기



3개가 연속된 경우 찾기

이 방법으로 비교 구문의 코드를 작성하면 다음과 같다.

```
#include <stdio.h>
int s[81];
int n, end=0;
bool same(int a, int b)
{
  int i;
  for(i=a; i<b; i++)
    if(s[i]!=s[i+b-a])
      break;
    return (a==b ? false:i==b);
}

int good(int m)
{
  for(int i=1; i<=m/2; i++)
    for(int j=1; j<=m-i; j++)
      if(same(j,j+i))
        return 0;
  return 1;
}

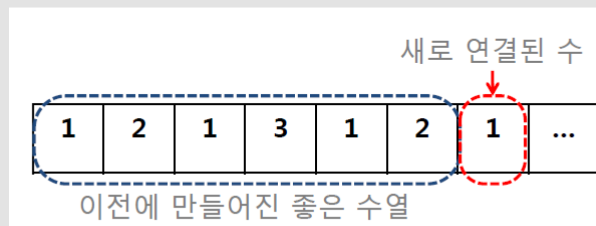
int main()
{
  scanf("%d", &n);
  solve(1);
  return 0;
}
```

위 비교함수를 이용하여 깊이우선탐색으로 전체탐색을 구현한 solve함수의 소스코드는 다음과 같다.

```
void solve(int k)
{
  if(end) return;
  if(k>n)
  {
    end=1;
    for(int i=1; i<=n; i++)
      printf("%d",s[i]);
    return;
  }
  for(int i=1; i<=3; i++)
  {
    s[k]=i;
    if(good(k)) solve(k+1);
    s[k] = 0;
  }
}
```

이 방법보다 조금 더 효율적인 방법을 생각해볼 수 있다. 만들어진 수열의 좋은 수열 여부를 판단할 때, 새로 붙은 수를 포함하는 것으로만 평가해 보는 것이다. 이 때 좋은 수열을 판단하는 방법은 다음과 같다.

1. 이전까지 만들어진 수열은 좋은 수열이다.
2. 새로운 수를 하나 붙인다.
3. 새로운 수를 포함한 것들만 좋은 수열인지 평가한다.



이와 같은 방법으로 새로 연결된 수만 비교하는 과정은 다음과 같다.



이 방법을 이용하면 좋은 수열 판단에 이용했던 함수 same과 good을 다음과 같이 수정할 수 있다.

```
#include <stdio.h>
int s[81];
int n, end=0;

bool same(int a, int b)
{
  int i;
  for(i=a; i<b; i++)
    if(s[i]!=s[i+b-a])
      break;
  return (a==b ? false:i==b);
}

int good(int m)
{
  int i, j;
  for(i=m-1, j=m; i>0; i-=2,j-=1)
    if(same(i,j))
      return 0;
  return 1;
}
```

문제 25

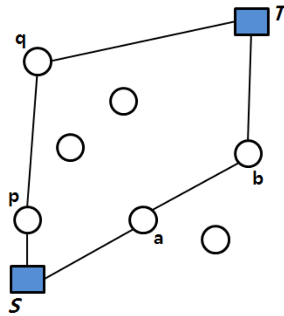
경비행기(S)

경비행기 독수리호가 출발지 S에서 목적지 T로 가능한 빠른 속도로 안전하게 이동하고자 한다. 이 때, 경비행기의 연료통의 크기를 정하는 것이 중요한 문제가 된다.

큰 연료통을 장착하면 중간에 내려서 급유를 받는 횟수가 적은 장점이 있지만 연료통의 무게로 인하여 속도가 느려지고, 안정성에도 문제가 있을 수 있다.

한편 작은 연료통을 장착하면 비행기의 속도가 빨라지는 장점이 있지만 중간에 내려서 급유를 받아야 하는 횟수가 많아지는 단점이 있다.

문제는 중간에 내려서 급유를 받는 횟수가 k 이하일 때 연료통의 최소 용량을 구하는 것이다. 아래 예를 보자.



위 그림은 S, T와 7개의 중간 비행장의 위치를 나타내고 있다. 위 예제에서 중간급유를 위한 착륙 허용 최대횟수 k = 2라면, S-a-b-T로 가는 항로가 S-p-q-T로 가는 항로보다 연료통이 작게 된다.

왜냐하면, S-p-q-T 항로에서 q-T의 길이가 매우 길어서 이 구간을 위해서는 상당히 큰 연료통이 필요하기 때문이다.

문제는 이와 같이 중간에 최대 k번 내려서 갈 수 있을 때 최소 연료통의 크기가 얼마인지를 결정하여 출력하는 것이다.

참고사항은 다음과 같다.

경비행기(S) (계속)

1) 모든 비행기는 두 지점 사이를 반드시 직선으로 날아간다. 거리의 단위는 km이며 연료의 단위는 ℓ(리터)이다. 1ℓ당 비행거리는 10km이고 연료 주입은 ℓ 단위로 한다.

2) 두 위치 간의 거리는 평면상의 거리이다. 예를 들면, 두 점 g = (2, 1)와 h = (37, 43) 간의 거리 d(g, h)는 sqrt{ (2-37)^2 + (1-43)^2 }= 54.671이고 50 < d(g, h) ≤ 60이므로 필요한 연료는 6ℓ가 된다.

3) 출발지 S의 좌표는 항상 (0, 0)이고 목적지 T의 좌표는 (10000,10000)으로 모든 입력 데이터에서 고정되어 있다.

4) 출발지와 목적지를 제외한 비행장의 수 n은 3 <= n <= 10이고 그 좌표값 (x, y)의 범위는 0 < x, y < 10,000인 정수이다. 그리고 0 <= k <= 1000이다.

입력

입력의 첫 줄에는 n과 k가 하나의 공백을 두고 주어진다. 그 다음 n개의 줄에는 각 비행장(급유지)의 정수좌표가 "x y"의 형식으로 주어진다.

출력

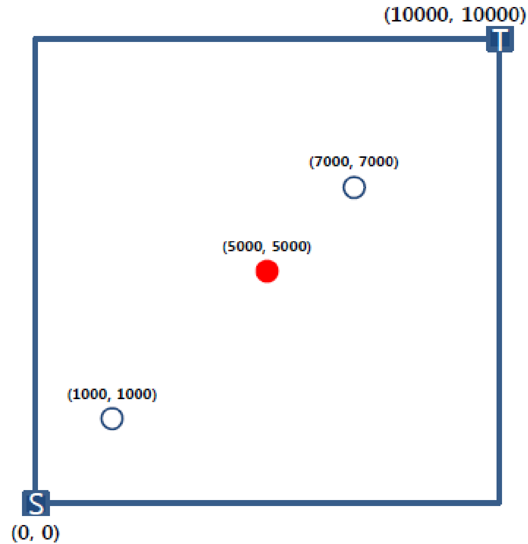
출력에는 S에서 T까지 k번 이하로 중간급유를 하여 갈 수 있는 항로에서의 최소 연료통 용량에 해당하는 정수를 출력한다.

입력 예	출력 예
10 1 10 1000 20 1000 30 1000 40 1000 5000 5000 1000 60 1000 70 1000 80 1000 90 7000 7000	708

출처: 한국정보올림피아드(2005 전국본선 고등부)

풀이

출발지(0,0)와 도착지(10000,10000)는 고정되어 있고, 연료 급유에 대한 조건이 급유지 수, 급유 횟수, 급유지 좌표로 주어졌을 때, 필요한 연료통의 최소 크기를 구해야 하는 문제이다.



그림에서 가능한 주유 횟수가 1번이라면, (5000,5000) 급유지에서 1번 주유하고 도착지에 도착할 수 있으며, 필요한 연료는 708리터이다.

시작점부터 (5000,5000) 급유지까지의 거리는 $\sqrt{5000^2+5000^2}=7071.067$ 이므로, 최소 708리터 필요하다.

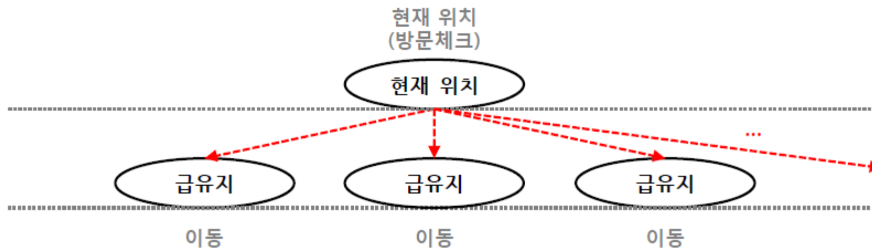
단순한 생각으로는 출발지에서 가능한 급유지를 선택해 가면서, 가능한 모든 경우를 판단해 볼 수도 있지만 이는 거의 불가능한 방법이다.

예를 들어 k번 급유가 가능하면, 1번 급유하고 도착할 수 있는 경우, 2번.. k-1번, k번 급유하고 도착할 수 있는 경우 중 가장 작은 연료통의 크기가 정답이 된다.

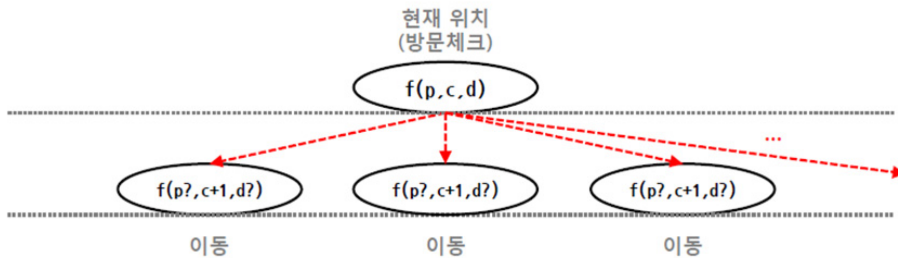
가장 간단한 문제 해결전략은 전체탐색법으로 가능한 모든 경우를 조사해 보는 방법이다. 이 방법은 시간이 많이 걸리는 단점이 있지만, 항상 정확한 해를 구할 수 있다는 장점

이 있다. 단, 적은 데이터에 대해서만 제한 시간 내에 정확한 결과를 얻을 수 있다.

현재 위치에서 이동 가능한 모든 지점을 확인한 후, 다시 그 위치에서, 지금까지 지나오지 않은 이동 가능한(체크하지 않은) 모든 다른 지점을 순차적으로 확인해 나갈 수 있고, 이렇게 이동 횟수가 k번 이하로 목적지까지 도착하는 경우까지 수행하면 모든 경우를 확인할 수 있다.



현재 급유지 번호를 p, 현재까지의 급유횟수를 c, 현재까지 이동하면서 가장 길었던 구간길이를 d라고 하면, 그림과 같이 선택해 나갈 수 있다. 주의 깊게 생각해야 하는 것은, 거치지 않았던 도시들에 대해서만 이동해 가야 한다는 점이다.



또한, 만약 k 번째 급유지에 도착했다면? 그 동안 지나왔던 가장 먼 거리와, 그 급유지에서 마지막 도착점까지의 거리 중 큰 거리가 그 경로를 타고 목적지에 도착하기 위해 필요한 가장 큰 거리가 된다. 그렇게 얻어진 “가능한 경로에서 가장 긴 구간들 중 가장 작은 값”이 주어진 문제의 답이 된다.

출발지를 0번째 급유지, 목적지를 마지막 급유지라고 생각할 수 있다.

경유한 급유지를 기록해 두기 위해 v[ ]을 사용하고, 각 급유지의 (x, y) 좌표를 저

장하기 위해 구조체 a[ ]로 정의할 수 있다. 또한 다음 급유지까지의 거리를 계산하기 위한 함수를 따로 만들고 활용할 수 있다.

또한, 두 급유지 사이의 직선거리를 계산하기 위해 제곱근(루트)를 구해야 하는데, 〈math.h〉의 sqrt(square root) 함수를 사용할 수 있다.

얻어진 거리를 10으로 나눈 후(1리터 당 10킬로미터를 움직일 수 있으므로..) 가장 가까우면 서 큰 정수(가능한 연료통 부피)로 바꾸기 위해서 ceil( ) 함수를 사용할 수도 있다.

예를 들어 얻어진 실수가 1.4.. 1.1… 1.9 등 1보다 큰 수인 경우보다 큰 정수인 2로 계산되어야 하는데 이때에 천장(ceil)을 의미하는 함수 ceil( )을 이용해 쉽게 계산할 수 있다. 반대되는 의미로는 바닥을 의미하는 floor( ) 함수도 있다.

위 방법으로 구현한 소스코드는 다음과 같다.

코드	참고

```
#include <stdio.h>
#include <math.h>

int n, k, v[11], dmin=200000000;

struct air {int x; int y;} a[11];
int max(int a, int b) { return a>b ? a:b;}

int dist(int p1, int p2)
{
  return
(a[p1].x-a[p2].x)*(a[p1].x-a[p2].x)+(a[p1].y-a[p2].y)*(a[p1].y-a[p2].y);
}

void f(int p, int c, int d)
{
  if(c==k)
  {
    d=max(d, dist(p,n+1));
    if(d<dmin) dmin=d;
    return;
  }
```

```
  for(int i=0; i<=n+1; i++)
    if(!v[i])
    {
      v[i]=1;
      f(i,c+1,max(d,dist(p,i)));
      v[i]=0;
    }
}

int main()
{
  scanf("%d %d", &n, &k);
  a[0].x=0, a[0].y=0;
  for(int i=1; i<=n; i++)
    scanf("%d %d", &a[i].x, &a[i].y);
  a[n+1].x=10000, a[n+1].y=10000;
  v[0]=1;
  f(0,0,0);
  printf("%.f\n", ceil(sqrt(dmin)/10));
  return 0;
}
```

문제 26

돌다리 건너기(S)

절대반지를 얻기 위하여 반지원정대가 출발한다. 원정대가 지나가야할 다리는 두 개의 인접한 돌다리로 구성되어 있다. 하나는 <악마의 돌다리>이고 다른 하나는 <천사의 돌다리>이다. 아래 그림 1은 길이가 6인 다리의 한 가지 모습을 보여준다.

그림에서 위의 가로줄은 <악마의 돌다리>를 표시하는 것이고 아래의 가로줄은 <천사의 돌다리>를 표시한다. 두 돌다리의 길이는 항상 동일하며, 각 칸의 문자는 해당 돌에 새겨진 문자를 나타낸다. 두 다리에 새겨진 각 문자는 {R, I, N, G, S} 중 하나이다.

출발	R	I	N	G	S	R	도착
	G	R	G	G	N	S	

그림 1

반지원정대가 소유하고 있는 마법의 두루마리에는 <악마의 돌다리>와 <천사의 돌다리>를 건너갈 때 반드시 순서대로 밟고 지나가야할 문자들이 적혀있다. 이 순서대로 지나가지 않으면 돌다리는 무너져, 반지원정대는 화산 속으로 떨어지게 된다. 다리를 건널 때 다음의 제한조건을 모두 만족하면서 건너야 한다.

(1) 왼쪽(출발지역)에서 오른쪽(도착지역)으로 다리를 지나가야 하며, 반드시 마법의 두루마리에 적힌 문자열의 순서대로 모두 밟고 지나가야 한다.
(2) 반드시 <악마의 돌다리>와 <천사의 돌다리>를 번갈아가면서 돌을 밟아야 한다. 단, 출발은 어떤 돌다리에서 시작해도 된다.
(3) 반드시 한 칸 이상 오른쪽으로 전진해야하며, 건너뛰는 칸의 수에는 상관이 없다.

만일 돌다리의 모양이 그림 1과 같고 두루마리의 문자열이 “RGS”라면 돌다리를 건너갈 수 있는 경우는 다음의 3가지뿐이다(아래 그림에서 큰 문자는 밟고 지나가는 돌다리를 나타낸다) .

출발	**R**	I	N	G	**S**	R	도착
	G	R	**G**	G	N	S	

출발	**R**	I	N	G	**S**	R	도착
	G	R	G	**G**	N	S	

출발	R	I	N	**G**	S	R	도착
	G	**R**	G	G	N	**S**	

돌다리 건너기(S) (계속)

아래의 세 방법은 실패한 방법이다.

출발	**R**	I	N	G	S	R	도착
	G	R	**G**	G	N	S	

출발	**R**	I	N	**G**	S	R	도착
	G	R	G	G	N	**S**	

출발	**R**	I	N	G	**S**	R	도착
	G	R	G	G	N	S	

왜냐하면 첫 번째는 문자열 "RGS"를 모두 밟고 지나가야 하는 조건 (1)을 만족하지 않으며, 두 번째는 번갈아가면서 돌을 밟아야 하는 조건 (2)를, 세 번째는 앞으로 전진을 하여야하는 조건 (3)을 만족하지 않기 때문이다.

마법의 두루마리에 적힌 문자열과 두 다리의 돌에 새겨진 문자열이 주어졌을 때, 돌다리를 통과할 수 있는 모든 가능한 방법의 수를 계산하는 프로그램을 작성하시오. 예를 들어, 그림 1의 경우는 통과하는 방법이 3가지가 있으므로 3을 출력해야 한다.

입력

첫째 줄에는 마법의 두루마리에 적힌 문자열(R, I, N, G, S로만 구성된)이 주어진다. 이 문자열의 길이는 최소 2, 최대 10이다. 그 다음 두 줄에는 각각 <악마의 돌다리>와 <천사의 돌다리>를 나타내는 같은 길이의 문자열이 주어진다. 그 길이는 5 이상, 20 이하이다.

출력

출력 파일에 마법의 두루마리에 적힌 문자열의 순서대로 다리를 건너갈 수 있는 방법의 수를 출력한다. 그러한 방법이 없으면 0을 출력한다. 모든 테스트 데이터에 대한 출력결과는 2^31 - 1 이하이다.

입력 예	출력 예
RGS RINGSR GRGGNS	3
RINGS SGNIRSGNIR GNIRSGNIRS	0
GG GGGGRRRR IIIIGGGG	16

출처: 한국정보올림피아드(2004 전국본선 고등부)

풀이

돌다리를 밟는 순서와 돌다리의 상태가 주어질 때, 돌다리를 건널 수 있는 모든 경우의 수를 찾는 문제이다. 만족해야 하는 조건은

- 두루마리에 적힌 순서대로 지나가야 한다.
- 악마의 돌다리와 천사의 돌다리를 번갈아 밟고 지나가야 한다.
- 출발은 어느 돌다리에서 하든지 상관없다.
- 반드시 오른쪽으로 한 칸 이상씩 전진해야 한다.
- 건너뛰는 칸의 수는 제한이 없다.

두루마리	R	G	S

악마의 돌다리	R	I	N	G	S	R	도착
천사의 돌다리	G	R	G	G	N	S	

RGS 순서로 돌다리를 밟고 건너가는 방법은 3가지뿐이다.

두루마리	R	G	S

경우 - ①

악마의 돌다리	R	I	N	G	S	R	도착
천사의 돌다리	G	R	G	G	N	S	

경우 - ②

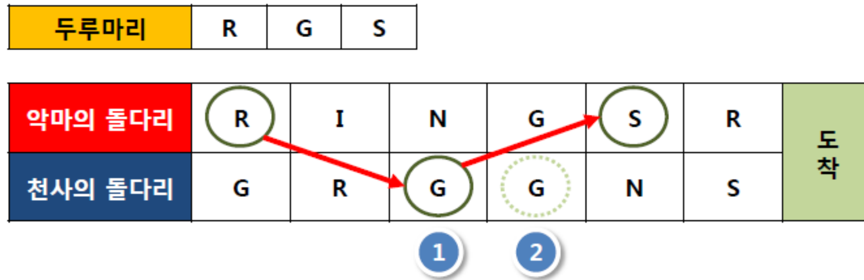
악마의 돌다리	R	I	N	G	S	R	도착
천사의 돌다리	G	R	G	G	N	S	

경우 – ③

악마의 돌다리	R	I	N	G	S	R	도착
천사의 돌다리	G	R	G	G	N	S	

문제 해결 전략을 만들기 위해 가장 단순한 과정을 생각해 볼 수 있다.

악마의 돌다리에서 시작해서 두루마리 순서대로 움직여서 밟아 간다고 생각하면,



악마의 돌다리에서 첫 번째 R을 선택하면, 반대편 천사의 돌다리에서 두 번째 G를 선택해야 한다. 그런데 천사의 돌다리에서 선택할 수 있는 G는 여러 가지가 있음을 생각할 수 있다.

따라서 먼저 어떤 돌다리에서 하나를 선택했다면, 반대편 돌다리에서 가능한 모든 경우에 대해서 각각 선택해보고, 그 다음 두루마리 순서로 그 반대편에서 다시 각각 선택해서, 도착할 때까지 가는 모든 경우를 세어보는 방법이 있다.

도착할 수 있는 경우는, 더 나아갈 수 있는 돌다리가 더 남았는데 두루마리에 남아있는 다음 문자가 더 없는 경우로 판단 할 수 있다.

두루마리에 남아있는 문자가 더 없다면, 가능한 한 가지 경우를 찾은 것이고, 남아있는 문자가 있는데, 돌다리를 모두 지나왔다면, 가능하지 않은 경우이다.

반대편 돌다리에서 가능한 모든 경우에 대해서, 확인할 수 있는 전체탐색법을 설계해 볼 수 있다. 이를 구현한 소스코드는 다음과 같다.

```c
#include <stdio.h>

char r[11];
char b1[21];
char b2[21];

int f(int s, int n, int rp)
{
  int i, c=0;
  if(r[rp]=='\0') return 1;
  if(s==1)
  {
    for(i=n; b1[i]!='\0'; i++)
      if(b1[i]==r[rp])
        c+=f(2, i+1, rp+1);
  }
  if(s==2)
  {
    for(i=n; b2[i]!='\0'; i++)
      if(b2[i] == r[rp])
        c+=f(1, i+1, rp+1);
   }
   return c;
}

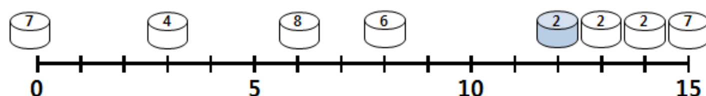
int main()
{
  scanf("%s %s %s", r, b1, b2);
  printf("%d\n", f(1,0,0)+f(2,0,0));
  return 0;
}
```

문제 27

공주 구하기(S)

유시 섬에서 한가롭게 소풍을 즐기던 다리오와 오렌지 공주.

다리오가 잠시 자리를 비운 사이에 못된 악당 후퍼가 공주를 데리고 도망가 버렸다. 다리오는 후퍼가 오렌지 공주를 숨겨놓은 후퍼 섬으로 여행을 떠난다.



유시 섬에서 후퍼 섬까지 가기 위해서는 중간에 있는 여러 개의 섬을 거쳐 가야 한다. 유시 섬과 후퍼 섬을 포함한 모든 섬들은 유시 섬과 후퍼 섬을 지나는 직선상에 있다. 위 그림에서, 섬들을 나타내는 동그라미 아래에 있는 눈금자가 각각의 섬이 유시 섬과 몇 km나 떨어져 있는지를 나타낸다.

가장 왼쪽에 있는 섬이 유시 섬이고, 가장 멀리 있는 후퍼 섬은 15km 떨어져 있다. 한 섬에서 다른 섬으로 건너가기 위해서는 섬마다 하나씩 있는 스프링 발판을 밟아 점프해야 한다. 이 스프링 발판은 내구성이 약해서 한 번 사용하면 부서져 버린다.

이 때문에, 시작점인 유시 섬을 제외한 모든 섬들은 두 번 이상 방문하면 안 된다. 스프링 발판들의 스프링의 세기는 모두 다르다. 섬을 나타내는 동그라미에 쓰여 있는 숫자는 스프링 발판을 딛고 점프했을 때 가장 멀리 도달할 수 있는 거리를 나타낸다.

가령, 유시 섬에서 7km 떨어져 있는 섬의 스프링 발판의 세기가 3이라면, 스프링 발판을 딛고 도달할 수 있는 섬은 유시 섬에서 4km 이상 10km 이하 떨어져 있는 섬들이다. 다리오는 공주를 구하기 위해 앞만 보고 질주한다. 공주를 구하기 전에는 스프링 발판을 밟고 후퍼 섬을 향해서만 점프한다.

하지만 공주를 구한 뒤에는 공주를 들쳐 업고 유시 섬을 향해서만 뒤도 돌아보지 않고 도망친다. 일부 스프링 발판은 내구도가 너무 약해서 공주를 들쳐 업은 상태에서는 발만 딛어도 부서져버리기도 한다.

그림에서 유시 섬에서 12km 떨어진 곳에 있는 회색으로 표시된 섬의 스프링 발판이 그 예이다.

공주 구하기(S) (계속)

이런 스프링 발판들은 공주를 구하러 후퍼 섬을 향해 갈 때에만 사용할 수 있다.

유시 섬과 후퍼 섬을 포함한 모든 섬들의 정보와 섬마다 하나씩 있는 스프링 발판의 정보가 주어질 때, 다리오가 유시 섬을 출발해 공주를 구하고 돌아오는 서로 다른 경로의 개수를 1000으로 나눈 나머지를 출력하는 프로그램을 작성하시오.

입력

첫째 줄에는 섬의 개수 n이 주어진다. n은 유시 섬과 후퍼 섬도 포함한다.

(3 <= n <= 20)

이어지는 n개의 줄에는 각각의 섬에 대한 정보가 한 줄에 하나씩 주어진다. 섬의 정보는 유시 섬과의 거리가 가까운 순으로 주어진다. 그러므로 첫 번째로 정보가 주어지는 섬은 항상 유시 섬이고, 마지막으로 정보가 주어지는 섬은 항상 후퍼 섬이다.

섬의 정보를 나타내는 각각의 줄에는 섬에 대한 정보를 표현하는 세 개의 정수가 빈칸을 사이에 두고 주어진다. 첫 번째 정수 p는 유시 섬과의 거리이다. 유시 섬에 대해서는 p는 0이고, 후퍼 섬에서 p값이 가장 크다. p값이 동일한 두 섬은 존재하지 않는다.

두 번째 정수 d는 스프링 발판의 세기, 즉 해당 섬에서 좌우로 얼마나 떨어진 섬까지 점프할 수 있는지를 나타내는 값이다. 세 번째 정수 g는 해당 섬의 스프링 발판을 오렌지 공주를 들쳐 업은 상태에서도 사용할 수 있는지를 나타내는 값이다.

1이면 오렌지 공주와 함께 이용할 수 있고, 0이면 이용할 수 없다. 후퍼 섬에서 이 값은 항상 1이다.

출력

첫째 줄에 유시 섬에서 출발해 오렌지 공주를 구해오는 총 경로의 수를 1,000으로 나눈 나머지를 출력한다.

입력 예	출력 예
8 0 7 1 3 4 1 6 8 1 8 6 1 12 2 0 13 2 1 14 2 1 15 7 1	6

출처: 한국정보올림피아드(2008 지역본선 중고등부)

풀이

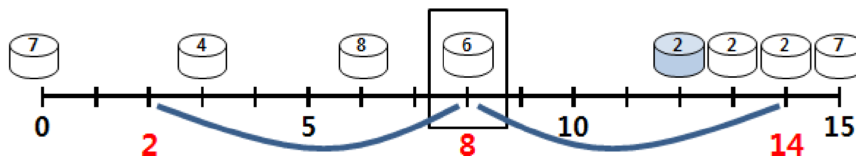
처음 위치에서 출발해서 마지막 위치에 도착한 후, 다시 처음 위치로 돌아오는 모든 경우의 수를 계산하고 그 결과를 이용해 답을 출력해야 하는 문제이다.

단, 어떤 발판을 밟게 되면 최대로 도달 가능한 다음 발판까지의 거리가 주어지며, 한 번 사용한 발판은 다시 사용하지 못한다.

또한, 마지막 위치에 도달한 후 다시 처음 위치로 돌아올 때에는 사용할 수 없는 특별한 발판이 존재한다.

문제에서는 마지막 위치까지 도달 할 수 있는 경우를, 각 발판에서 점프할 수 있는 최대 크기와 오는데 사용할 수 없는 발판 정보를 이용해 효과적으로 구해내는 문제 해결 방법을 찾아내는 것이 핵심이 된다.

예를 들어 8번 위치에 있는 섬의 발판을 이용하는 경우, 그 발판의 강도가 6이므로 다음 섬은 2km이상 14km이하 떨어져 있는 섬들 중 하나로 이동할 수 있다.



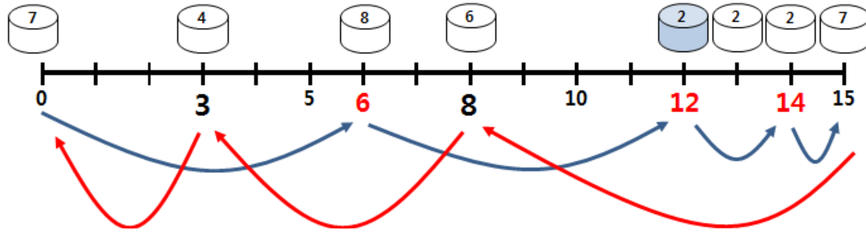
즉, 8번 위치에 있는 발판을 밟는 경우 3, 6, 12, 13, 14번 섬으로 이동 가능하다.

처음 0위치에서 7만큼 이동 가능한 발판은 3, 6에 위치가 가능하고, 만약 6위치의 발판을 선택 하게 되면 그 다음에 8만큼 이동 가능한 8, 12, 13, 14 위치의 한 섬을 선택할 수 있다.

위와 같은 과정을 반복적으로 실행해, 원래의 처음 위치로 돌아오는 모든 경우를 찾아야 한다.

예를 들어 첫 번째 섬에서 출발해 마지막 섬에서 공주를 구해 돌아오는 한 가지 경로에 대해서, "발판위치(발판강도)"를 이용해 아래와 같이 표현할 수 있다.

0(7) → 6(8) → 12(2) → 14(2) → 15(7) → 8(6) → 3(4) → 0

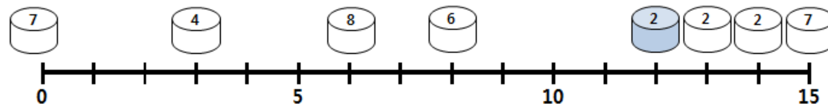


이런 방식으로 가능한 모든 경우의 수를 효과적으로 구해야 한다.

문제 해결전략은 두 사람 A, B가 시작 섬에서 마지막 섬으로 동시에 출발한다고 생각하고, A, B가 모두 마지막 섬에 도착할 수 있는지를 판단하는 문제로 생각하는 것이다.

A와 B는 시작 섬과 마지막 섬을 제외하고는 같은 섬을 밟으면 안 되는데, 그렇게 생각하면 A가 마지막 섬에 도착한 후, B가 돌아오는 경로로 A가 함께 돌아올 수 있는 1가지 경우가 되기 때문이다. 따라서 시작 섬과 마지막 섬을 제외하고는 A와 B가 움직이는 섬이 겹치지 않아야 한다.

A, B가 마지막 섬으로 도착할 수 있는 방법은 다음과 같이 나타낼 수 있다.



A : 0(7) – 6(8) – 12(2) – 14(2) – 15(7)

B : 0(7) – 3(4) – 8(6) – 15(7)

섬의 위치를 $p(i)$, 섬의 강도를 $d(i)$로 표시하면 다음 섬으로 이동 가능한 조건은 $p(i+1) <= p(i)+d(i)$이고, 이 조건을 만족하는 경우의 조합에 대해 A, B의 도착여부를 확인해 가능한 경우의 수를 찾아낼 수 있다.

위와 같이 이동한 경우 A, B가 모두 마지막 섬에 도착 가능하므로, A가 처음 위치에서 시작해 마지막에 도착하고, 다시 A와 B가 함께 처음 위치로 갈 수 있는 1가지의 경우를 찾

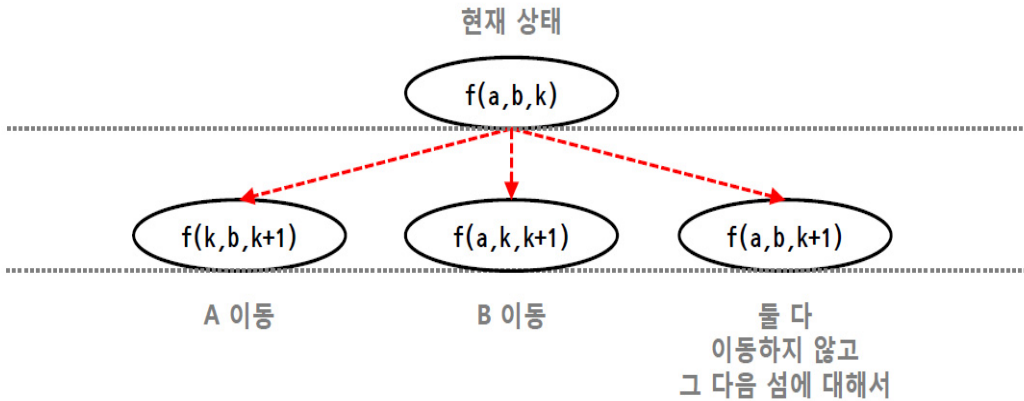
아낸 것과 같다.

어떤 섬 i 를 A, B 모두 밟은 상태에서 마지막 섬에 도착한 상태이면 1가지 경우를 더 카운팅하고, 그렇지 않으면 다음 섬을 선택하는 3가지의 경우를 생각할 수 있다.

다음 섬은 둘 중 하나만 밟거나, 둘 다 사용하지 않아야 한다. 따라서 가능한 3가지 경우는

- A만 i 다음 섬으로 이동하는 경우
- B만 i 다음 섬으로 이동하는 경우
- A, B 모두가 i 다음 섬을 사용하지 않는 경우

현재 A의 위치 a, B의 위치 b, 다음 섬의 위치를 k이라고 하면, f(a,b,k)은 현재 A가 a, B가 b의 위치에 있고 다음에 가능한 섬이 k인 것을 의미한다. 따라서 다음 상태로 가능한 경우를 아래와 같이 트리로 표현할 수 있다.



A가 이동하는 경우는 $p(n)$ <= $p(a)+d(a)$를 만족해야 하고 B가 이동하는 경우는 조심해야 하는데, 현재 위치에서 다음 섬으로 뛸 수 있는지를 확인하는 것이 아니라, 그 다음 섬(n)에서 지금 위치 $p(b)$로 뛰어 올 수 있는지를 생각해야 한다는 것이다.

따라서 $p(n)$ <= $p(b)+d(n)$을 만족하고 다음 섬이 회색이 아니어야 한다는 것이다. 섬의 위치를 $p[]$, 각 섬의 강도를 $d[]$, 회색 섬 여부를 $g[]$에 저장하고 문제 해결을 시도할 수 있다.

이를 구현한 소스코드는 다음과 같다.

```
#include <stdio.h>
int n, m=1000;
int p[500],d[500],g[500];
bool ca(int a, int k)
{
   return p[k]<=p[a]+d[a];
}

bool cb(int b, int k)
{
   return (p[k]<=p[b]+d[k])&&g[k];
}

int f(int a, int b, int k)
{
  int c=0;
  if(k==n-1)
  {
    if(ca(a,k)&&cb(b,k)) c=1;
    else c=0;
  }
  else
  {
    if(ca(a,k)) c+=f(k,b,k+1)%m;
    if(cb(b,k)) c+=f(a,k,k+1)%m;
    c+=f(a,b,k+1)%m;
  }
  return c;
}

int main()
{
  scanf("%d", &n);
  for(int i=0; i<n; i++)
    scanf("%d %d %d", &p[i], &d[i], &g[i]);
  printf("%d\n", f(0,0,1));
  return 0;
}
```

문제 28

소방차

직선 위에 여러 개의 소방펌프가 있다. 여러 대의 소방차가 물을 채우기 위해서 급하게 이 직선 위에 정차했다. 펌프의 수는 소방차의 수 보다 크거나 같다. 그림에는 두 대의 소방차 (위치는 27과 73)가 세 개의 펌프 (위치는 12, 50, 81) 사이에 정차한 것을 보여주고 있다.



소방차에서 물을 채우기 위해 펌프와 소방차 호스를 연결한다. 시간을 절약하기 위해서 모든 소방차에 동시에 물을 채우려 한다.

하나의 펌프에는 하나의 소방차만 연결될 수 있다. 사용하는 호스의 길이는 펌프와 소방차 사이의 거리이다. 그림의 경우, 첫 번째 소방차는 첫 번째 펌프의 연결하고 (호스 길이 15) 두 번째 소방차는 세 번째 펌프와 연결하면 (호스 길이는 8) 사용하는 호스 길이의 합은 15+8 = 23이다.

이렇게 하는 것이 호스 길이의 합을 최소로 한다. 펌프들의 위치와 소방차들의 위치가 주어질 때 호스 길이의 합을 최소로 하면서 펌프들을 소방차들에 연결하는 방법을 구하는 프로그램을 작성하시오.

입력

첫째 줄에는 펌프의 수를 나타내는 정수 p와 소방차의 수를 나타내는 f가 주어진다. $1 \le p \le 11$이고 $1 \le f \le 10$이며 $p \ge f$이다. 둘째 줄에는 펌프들의 위치를 나타내는 서로 다른 p개의 정수가 오름차순으로 주어진다. 셋째 줄에는 소방차들의 위치를 나타내는 서로 다른 f개의 정수가 오름차순으로 주어진다. 펌프와 소방차가 같은 위치에 있을 수도 있다. 주어진 정수는 모두 1,000,000 이하의 양수이다.

출력

사용하는 호스 길이의 합을 출력한다. 출력 결과는 2^31 - 1을 넘지 않는다.

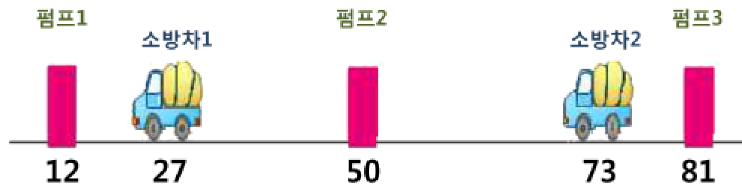
입력 예	출력 예
3 2 12 50 81 27 73	23

출처: 한국정보올림피아드(2005 전국본선 고등부)

풀이

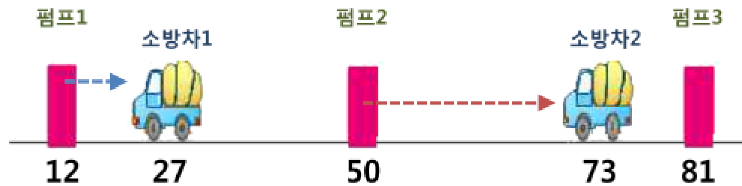
소방차에 물을 공급할 수 있는 펌프들의 위치와 소방차들의 위치가 주어졌을 때, 모든 소방차를 펌프들에 연결하기 위한 최소 호스 길이의 합을 찾아내는 문제이다.

예시를 통해 상황을 살펴보면, 소방차를 펌프에 연결하는 방법은 모두 6가지이다.

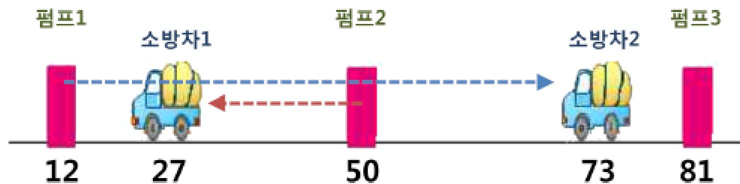


p개의 펌프 중에서 f개의 펌프를 골라, f개의 각 소방차들에게 할당하는 모든 경우가 가능한데, 예시 그림에서는 3개의 펌프 중에서 2개를 고르는 경우의 3가지가 가능하고, 각 펌프에 대해서 소방차1 또는 소방차2가 가능하기 때문에 총 6가지 경우가 가능하다.

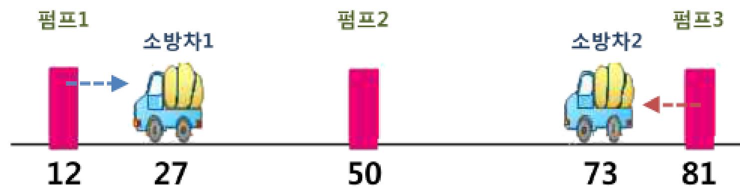
펌프1, 펌프2를 고른 경우, 펌프1-소방차1, 펌프2-소방차2 의 조합이 가능하다.



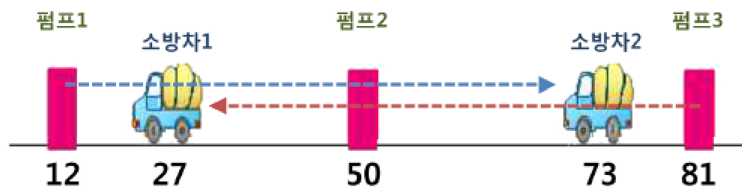
호스 길이의 합 = (27−12) + (73−50) = 15+23 = 38



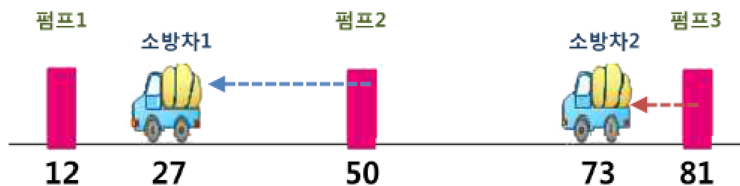
호스 길이의 합 = (73−12) + (50−27) = 61+23 = 84



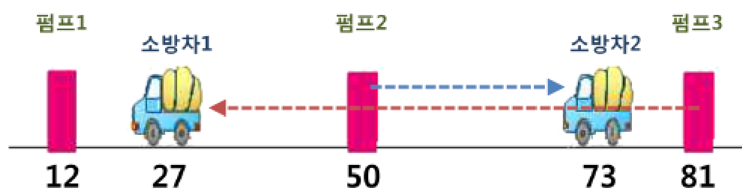
호스 길이의 합 = (27−12) + (81−73) = 15+8 = 23



호스 길이의 합 = (73−12) + (81−27) = 61+54 = 115



호스 길이의 합 = (50−27) + (81−73) = 23+8 = 31

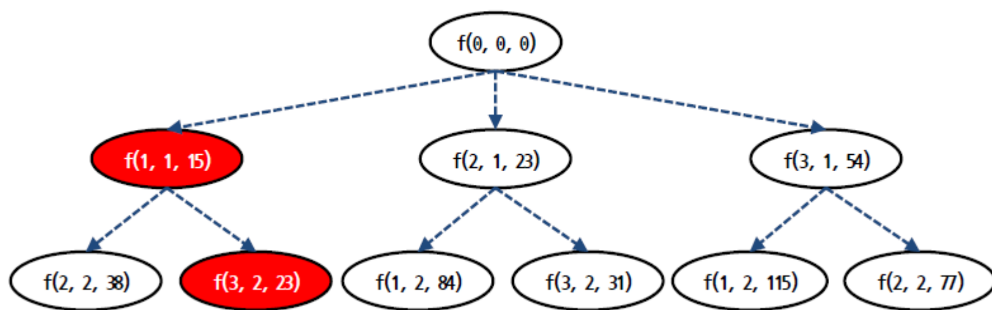


호스 길이의 합 = (81−27) + (73−50) = 54+23 = 77

가능한 모든 경우에 대해서 탐색해 보는 문제 해결 전략을 적용해 볼 수 있다.

펌프번호 p, 소방차 번호 f, 누적 호스 길이 h 라고 하면, 가능한 상태들을 상태 트리로 만들어 가는 백트랙킹 방법을 만들 수 있다.

f(0,0,0) 은 아무 소방차도 연결되지 않은 상태, f(1,1,15) 는 1번 소방차가 1번 펌프 연결되어 누적 호스 길이가 15가 된 상태를 의미한다.



위 방법을 소스코드로 구현한 결과는 다음과 같다.

코드	참고

```
#include <stdio.h>
int p, f, pp[100001], fp[100000], pv[100001],mh=0x7fffffff;

int min(int a, int b)
{
   return a>b ? b:a;
}

int abs(int a)
{
   return a>0 ? a:-a;
}

void g(int pt, int ft, int h)
{
  if(ft==f)
```

줄	코드	참고

```
  {
    mh=min(mh, h);
    return;
  }
  for(int i=1; i<=p; i++)
  {
    if(pv[i]==0)
    {
      pv[i]=1;
      h+=abs(pp[i]-fp[ft+1]);
      g(i, ft+1, h);
      h-=abs(pp[i]-fp[ft+1]);
      pv[i]=0;
    }
  }
}

int main()
{
  int i;
  scanf("%d %d", &p, &f);
  for(i=1; i<=p; i++)
    scanf("%d", &pp[i]);
  for(i=1; i<=f; i++)
    scanf("%d", &fp[i]);
  g(0,0,0);
  printf("%d\n", mh);
  return 0;
}
```

6 탐색공간의 배제

전체탐색법은 대부분의 경우 해를 구할 수 있는 알고리즘이다. 하지만 실행시간이 너무 길어 제한 시간 내에 문제를 해결할 수 없는 경우가 많다. 탐색공간의 배제는 전체탐색 알고리즘을 구현하는 데 있어서 더 이상 탐색하지 않더라도 해를 구하는 데 문제가 없는 부분을 판단하여 이 부분에 대해서 탐색을 하지 않으므로 탐색의 효율을 높이고자 하는 방법이다.

탐색공간의 배제는 전체탐색에서 불필요한 탐색공간을 탐색하지 않음으로써 알고리즘의 효율을 향상시킨다. 이와 같이 탐색공간을 배제하는 방법은 다양하며 가장 기본 전략은 전체탐색설계와 같이 탐색으로 시작하여 모든 공간을 탐색하는 것이 아니라 일정한 조건을 두어 탐색영역을 배제하는 것이다.

배제되는 탐색공간의 크기에 따라 알고리즘의 성능의 향상 폭이 달라진다. 하지만 잘못 설계를 하여, 해가 있는 상태를 배제하면 해를 구할 수 없는 경우가 발생한다. 따라서 탐색영역을 배제할 때는 엄밀한 수학적 접근이 필요하다.

이 설계방법은 탐색영역을 배제하는 방법에 따라서 수학적 배제, 경험적 배제, 구조적 배제로 나눌 수 있다. 각 방법에 대해서 자세히 알아보자.

가. 수학적 배제를 이용한 설계

탐색 공간 중 배제할 영역을 수학적 증명으로 결정하는 방법으로는 이분탐색 알고리즘이 있다. 이는 일종의 수학적 배제를 이용하여 탐색공간을 줄여나가는 알고리즘 설계방법이라고 할 수 있다.

오름차순으로 정렬된 상태의 이분탐색에서 현재 탐색한 값이 목표하는 값보다 작다면, 현재 탐색 위치의 왼쪽 영역에는 해가 존재할 가능성이 없다. 이는 수학적으로 쉽게 증명할 수 있다.

따라서 왼쪽 영역에 대해서는 탐색할 필요가 없음을 알 수 있다. 그러므로 다음 탐색영

역은 이를 배제하고 오른쪽 영역만 탐색하는 방법이다. 이와 같이 수학적으로 탐색할 필요가 없음이 증명된 공간들을 배제해 나가며 탐색하는 것과 같은 접근법이 수학적 배제를 이용한 방법이라고 할 수 있다.

수학적 배제로 알고리즘을 설계할 경우, 공간을 배제할 원리를 수학적으로 증명한 후, 이 방법을 반복적으로 해를 찾을 때까지 적용해 나가며 해를 찾는다. 탐색공간에서 선택 배제된 부분은 수학적으로 탐색할 필요가 없으므로, 일반적으로 탐색법이긴 하지만 백트랙 없이 선형으로 진행되는 경우가 많다.

수학적으로 공간을 배제해 나가는 이 방법은 일종의 탐욕법(greedy)이라고 할 수 있으며, 엄밀하게 수학적으로 증명을 하기 때문에 수학적 탐욕법(mathematical greedy)라고 할 수 있다.

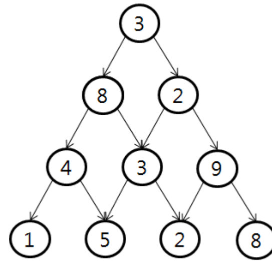
하지만 수학적 증명 없이, 직관적으로 현재 상태만으로 잘못된 판단을 하게 되면 올바른 해를 구할 수 없는 가능성을 가지는 단순 탐욕법이 될 수 있으므로 주의해야 한다. 하지만 단순 탐욕법의 경우에도 다양한 응용법이 있으므로 다음에 다루도록 한다.

다음 예는 루트 정점에서 출발하여 각 정점의 값을 누적하며 마지막 정점까지의 합을 최대화하는 최적화문제이다. 이 문제의 목적은 값을 최대화 하는 것이므로 다음 [영역배제의 규칙]을 적용하여 탐색 영역을 배제해 나가보자.

[영역배제의 규칙]

현재 상태에서 다음으로 탐색할 수 있는 정점들 중 더 높은 점수가 있는 정점으로 탐색한다.
(즉, 더 작은 점수가 있는 정점의 영역을 배제한다.)

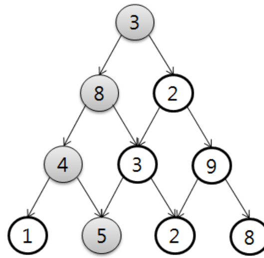
이 규칙은 수학적으로 설득력이 있어 보인다. 왜냐하면 값을 최대화하기 위해서는 작은 값보다는 큰 값이 이득이 되기 때문이다. 하지만 엄밀한 수학적 증명은 하지 않았다. 이 방법으로 탐색을 진행하는 과정은 다음과 같다.



최댓값 구하기

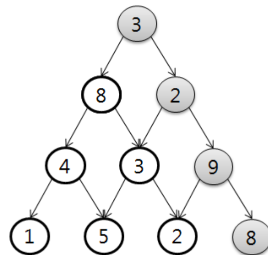
처음 출발점이 루트이므로 루트에 적힌 3점을 획득하여 현재 점수는 3점이다.

다음으로 이동할 수 있는 정점들은 왼쪽 아래로 연결된 8점이 기록된 정점과 오른쪽 아래로 연결된 2점이 기록된 정점의 2가지이다. 이 상태에서 [영역배제의 규칙]을 적용하여 값이 더 큰 8점이 기록된 정점을 선택하고 2점이 기록된 정점을 배제하고 진행한다.



잘못된 수학적 배제

이 규칙을 적용하여 마지막까지 탐색한 결과는 위 그림에서 구한 해는 $3-8-4-5$를 선택하게 되며 이 때 얻은 점수는 $3+8+4+5=20$ 이 된다. 과연 20점 이상을 획득할 수 있는 경로는 존재하지 않을까? 다음 그림을 보자.



최댓값 구하기의 최적해

위 결과를 보면 알 수 있듯이 $3+2+9+8=22$ 의 경로가 존재하며 앞에서 영역을 배제했던 규칙이 잘못됐음을 알 수 있다.

수학적 배제는 엄밀한 수학적 접근 없이, 단순히 직관적으로 배제의 규칙을 결정하면 최적해를 구할 수 있음을 보장할 수 없다. 하지만 구현이 간단하며, 일반적으로 최적해와의 차이가 크지 않은 해를 구할 수 있다는 장점을 이용하여 다른 설계법에 응용할 수 있으므로 나중에 다시 살펴보기로 하자.

주어진 문제들을 통하여 수학적 배제 방법으로 알고리즘을 설계해보자.

문제 1

약수의 합

한 정수 n을 입력받는다.

1부터 n의 자연수들 중 n 약수의 합을 구하는 프로그램을 작성하시오.

예를 들어 n이 10이라면

10의 약수는 1, 2, 5, 10이므로 구하고자 하는 값은 1 + 2 + 5 + 10을 더한 18이 된다.

입력

첫 번째 줄에 정수 n이 입력된다.
(단, 1 <= n <= 10,000,000,000(100억))

출력

n의 약수의 합을 출력한다.

입력 예	출력 예
10	18

풀이

이 문제는 앞에서 다루었던 약수의 합 문제와 동일한 문제이다. 차이점은 앞의 문제가 입력값의 정의역이 100,000이었던 것에 반해, 이 문제에서는 입력값이 100억으로 커졌다는 것이다.

앞의 문제에서 작성했던 풀이는 다음과 같다.

줄	코드	참고
1	`#include <stdio.h>`	
2	`int n;`	
3		
4	`int solve()`	
5	`{`	
6	`  int ans=0;`	
7	`  for(int i=1; i<=n; i++)`	
8	`    if(n%i==0)`	
9	`      ans+=i;`	
10	`  return ans;`	
11	`}`	
12		
13	`int main()`	
14	`{`	
15	`  scanf("%d", &n);`	
16	`  printf("%d\n", solve());`	
17	`  return 0;`	
18	`}`	

이 소스코드는 1부터 n까지의 모든 원소들을 탐색하여, 탐색 대상인 수 i가 n의 약수라면 취하는 방식으로 진행된다. 따라서 계산량은 $O(n)$이다.

이번 문제는 n의 최댓값이 100억이므로 이 방법으로는 너무 많은 시간이 걸린다. 따라서 탐색영역을 배제해야 할 필요가 있다.

먼저 간단한 수학적인 원리들을 생각해보자. 먼저 다음 정리를 이용하자.

모든 자연수 n에 대하여 1과 n은 항상 n의 약수이다.

이 원리를 이용하면 위 소스코드의 8행의 탐색범위를 줄여서 다음과 같이 표현할 수 있다.

줄	코드	참고
7 8 9	`for(int i=2; i<n; i++)` `  if(n%i==0)` `    ans+=i;`	

원래 소스코드보다 탐색공간이 줄어들긴 했으나 효율을 높이기에는 너무 미미하기 때문에 효율향상을 느낄 수 없다. 하지만 위 아이디어를 조금 응용하면 탐색공간을 많이 줄일 수 있다.

위 아이디어를 응용하기 위해서 다음 원리를 적용할 수 있다.

> 모든 자연수 n에 대하여,
>
> 2 이상 n 미만의 자연수들 중 가장 큰 n의 약수는 $\frac{n}{2}$를 넘지 않는다.

이 원리를 적용하면 다음과 같이 탐색영역을 줄일 수 있다.

줄	코드	참고
7 8 9	`for(int i=2; i<=n/2; i++)` `  if(n%i==0)` `    ans+=i;`	

이 알고리즘은 탐색영역이 처음의 소스코드의 반이하로 줄어든 것이다. 따라서 실행시간은 2배 이상 빨라질 것을 예상할 수 있다. 하지만 7행의 반복문이 한 번 실행될 때마다 $\frac{n}{2}$을 계산하기 위하여 나누기 연산을 하므로 다음과 같이 효율을 높이도록 바꿀 수 있다.

줄	코드	참고
7 8 9	`for(int i=2, bound=n/2; i<=bound; i++)` `  if(n%i==0)` `    ans+=i;`	

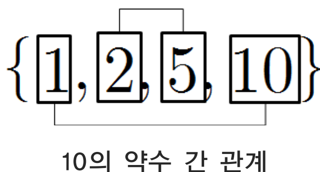
이와 같이 수정하면 나누기 연산을 반복횟수에 관계없이 한 번만 하므로 효율을 높일 수 있다. 하지만 컴파일러들은 최적화 관련 옵션 설정에 의해 이렇게 직접적으로 관계없는 연산을 반복문 외부로 빼내지 않더라도 자동으로 처리되는 경우도 있다.

수학적인 아이디어로 탐색 영역을 반 정도 줄였지만 아직도 매우 큰 입력 값을 처리하기에는 시간이 너무 오래 걸린다. 탐색 공간을 더 배제할 수 있는 아이디어를 생각해보자.

임의의 자연수 n의 약수들 중 두 약수의 곱은 n되는 약수 a와 약수 b는 반드시 존재한다. 단, n이 완전제곱수일 경우에는 약수 a와 약수 b가 같을 수 있다. 자연수 10의 약수를 통해서 알아보자. 자연수 10의 약수의 개수는 4개이며 다음과 같다.

$$\{1, 2, 5, 10\}$$

위 집합을 살펴보면 아래와 같은 관계를 찾을 수 있다.

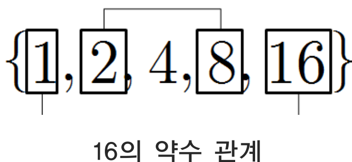


10의 약수 간 관계

위 그림에서 알 수 있듯이 1과 10의 곱은 10이고 2와 5의 곱은 10이다. 약수의 개수를 c개라고 하고, d_i를 n의 약수 중 i번째 약수라 하면 다음과 같은 식이 성립한다.

$$n = d_k \times d_{c-k+1}$$

즉, k번째 원소와 $c-k+1$번째 원소의 곱은 항상 n이다. 이 원리를 적용하면 10의 약수를 구할 때, 1과 2만 탐색하면 5와 10을 알 수 있으므로 모든 약수를 구할 수 있다. 단, n이 완전제곱수 일 경우에는 약수의 개수가 홀수 이므로 d_k번째 원소와 d_{c-k+1}번째 원소가 같을 경우가 한 건 존재한다. 완전제곱수인 16의 약수를 살펴보자.



16의 약수 관계

위 그림에서 알 수 있듯이 완전제곱수인 경우에는 $\left\lceil \frac{c}{2} \right\rceil$번째 원소는 짝이 없다. 따라서 $d_{\left\lceil \frac{c}{2} \right\rceil} \times d_{\left\lceil \frac{c}{2} \right\rceil} = n$이 된다. 즉 4와 4를 곱하여 16을 만들 수 있다.

이 원리를 적용하면 최악의 경우 2부터 $\sqrt{n}$까지만 탐색하면 모든 약수를 알 수 있다. 즉 100의 모든 약수를 구하려면 2부터 10까지만 조사해 보면 된다.

$$\{2,3,4,5,6,7,8,9,10\}$$

이 수들 중 10의 약수인 것만 찾아보면 다음과 같다.

$$\{2,4,5,10\}$$

위 약수들을 이용하여 짝을 찾아서 정리하면 다음과 같다.

$$\{2,4,5,10,20,25,50\}$$

이 된다. 여기에 1과 100은 당연히 100의 약수이므로 문제의 해는 다음과 같다.

$$1+2+4+5+10+20+25+50+100=217$$

탐색영역을 [2, $\sqrt{n}$] 로 설정할 때 일반적으로 다음과 같이 프로그램을 작성한다.

```
for( i = 1 ; i <= sqrt(n) ; i++ )
```

하지만 위와 같이 코딩하기 위해서는 sqrt 함수를 사용하기 위해서는 math.h를 추가적으로 include해야 하며, 반복문 내에서 매번 호출되는 sqrt 함수의 실행시간도 무시할 수 없기 때문에 더 효율적인 방법을 생각할 필요가 있다. 다음 부등식을 보자.

($i < \sqrt{n}$) 의 양변을 제곱하면 ($i^2 < n$)

이 방법을 이용하면 반복문을 다음과 같이 수정하여 사용할 수 있다.

```
for( i = 1; i*i <= n;  i++ )
```

이와 같이 간단한 수학적인 아이디어를 활용하면 효율적인 소스코드를 작성할 수 있으므로 항상 이런 아이디어를 활용할 수 있도록 익혀두자.

이처럼 탐욕적인 방법을 이용하면 큰 범위의 수도 컴퓨터 없이 쉽게 계산할 수 있다. 그

런데 이 방법을 프로그래밍으로 표현하기 위해서 주의할 점이 있다.

입력값 n이 100억 이기 때문에 자료형 int로는 이 값을 처리할 수 없다. 따라서 64bit형 정수인 long long int형을 활용해야 된다. 이 방법을 알고리즘으로 표현하면 다음과 같다.

코드	참고

```
#include <stdio.h>
long long int n;
long long int solve()
{
  long long int i, ans = 0;
  for(i=1; i*i<n; i++)
    if(n%i==0)
      ans+=(i+n/i);
  if(i*i==n)
    ans += i;
  return ans;
}
int main()
{
  scanf("%lld", &n);
  printf("%lld\n", solve());
  return 0;
}
```

문제 2

소수 구하기(S)

한 정수 n을 입력받는다.

n번째로 큰 소수를 구하여 출력한다.

예를 들어 n이 5라면

자연수들 중 소수는 2, 3, 5, 7, 11, 13, ...이므로 구하고자 하는 5번째 소수는 11이 된다.

입력

첫 번째 줄에 정수 n이 입력된다.
(단, 1 <= n <= 100,000)

출력

n 이하의 소수들의 합을 구하여 출력한다.

입력 예	출력 예
5	11
77	389

풀이

일반적으로 소수를 구하는 방법은 약수가 2개라는 성질을 이용하는 경우가 많다. 이 성질을 이용하여 임의의 정수 k가 소수인지 판단하는 알고리즘을 다음과 같이 만들 수 있다.

```
bool isPrime(int k)
{
  int cnt = 0;
  for(int i=1; i<=k; i++)
    if(k%i==0) cnt++;
  return cnt==2;
}
```

이 방법은 계산량이 $O(n)$이므로 효율이 좋지 않다. 결국 k번째 소수를 구하는 알고리즘은 $O(nk)$정도의 계산량이 요구되므로 원하는 시간 내에 답을 구하지 못할 가능성이 크다. 효율을 높이기 위해서는 탐색공간의 배제가 필요하다. 어떤 아이디어로 탐색공간을 줄일 수 있을까?

먼저 위 함수는 소수인지 판단하는 함수이며, 소수가 아니라면 약수가 몇 개이건 합성수인 것은 변함이 없으므로, 약수가 2개를 초과한다면 더 이상 탐색할 필요가 없다. 따라서 다음과 같이 isPrime 함수를 수정하여 탐색공간을 줄일 수 있다.

```
bool isPrime(int k)
{
  int cnt=0;
  for(int i=1; i<=k; i++)
  {
    if(k%i == 0) cnt++;
    if(cnt>2) break;
  }
   return cnt==2;
}
```

이와 같이 처리하면 대부분의 합성수는 매우 빠른 시간 내에 소수가 아님을 판정할 수 있다. 그리고 위 알고리즘을 다음과 같이 표현해도 된다. 각자 코딩스타일에 맞추어 원하는 방법을 익힐 수 있도록 한다.

줄	코드	참고
1	`bool isPrime(int k)`	
2	`{`	
3	`  int cnt=0;`	
4	`  for(int i=1; i<=k && cnt<=2; i++)`	
5	`    if(k%i == 0) cnt++;`	
6	`  return cnt==2;`	
7	`}`	

이번에 소스코드는 4행의 반복문의 반복조건을 바꾸어 처리하고 있다. 이렇게 하여 합성수를 빠르게 검사할 수 있지만 결국은 k번째 소수를 찾는 것이 목적이므로 소수를 검사할 때는 여전히 많은 시간이 걸린다. 소수를 보다 빠르게 검사할 수 있는 방법은 무엇일까?

다음 명제를 생각해보자.

> 임의의 자연수 n이 소수라면 n의 약수는 1과 n만 존재한다.

위 명제를 조금 변경하면 다음과 같은 원리를 생각할 수 있다.

> 임의의 자연수 n이 소수라면 구간 $[2,\ n]$에서 약수는 존재하지 않는다.

따라서 소수 판정 알고리즘을 다음과 같이 줄일 수 있다.

줄	코드	참고
1	`bool isPrime(int k)`	
2	`{`	
3	`  int cnt=0;`	
4	`  for(int i=2; i<k; i++)`	
5	`     if(k%i == 0) return false;`	
6	`  return true;`	
7	`}`	

이 방법도 합성수는 매우 빠르게 판정할 수 있지만 소수 판정은 시간이 많이 걸리는 단점이 있다. 하지만 이 방법으로부터 소수를 매우 빠르게 판정할 수 있는 방법을 만들 수 있다.

왜냐하면 주어진 범위에서 약수가 없어야 하므로, 약수의 존재성만 파악하면 된다. 약수의 존재성을 파악하기 위해서 모든 범위를 검사할 필요는 없다. 앞서 약수 문제에서 다루었던 것과 같이 n의 약수를 구하기 위해서 탐색을 $\sqrt{n}$까지만 탐색하면 된다. 소수 판정에서도 이 원리를 그대로 적용할 수 있다. 이 원리를 적용하여 소수 판정 알고리즘을 완성하면 다음과 같다.

```
bool isPrime(int k)
{
  int cnt = 0;
  for(int i=2; i*i<=k; i++)
    if(k%i == 0) return false;
  return true;
}
```

이 알고리즘은 매우 빠른 시간에 소수를 판정할 수 있다. $O(n^{\frac{1}{2}})$ 즉, $O(\sqrt{n})$으로 처리할 수 있다. 이 방법보다 더 빠른 방법이 있다. "에라토스테네스의 체"라는 방법을 이용하면 더 빠른 시간에 k번째 소수를 구할 수 있다. "에라토스테네스의 체"는 다음과 같은 단계를 거쳐 소수를 구한다.

준비. 2부터 n까지 차례로 숫자를 쓰고, 2부터 탐색을 시작한다.
1단계. 현재 탐색 중인 수가 지워지지 않았으면 그 수는 소수이다.
2단계. 1단계에서 그 수가 소수이면 그 수의 배수를 모두 지운다.
3단계. 만약 아직 탐색이 끝나지 않았으면 다음 수를 탐색할 준비를 하고 1단계로 간다.
4단계. 지워지지 않은 모든 수는 소수, 지워진 수는 합성수이다.

이 "에라토스테네스의 체"를 적절히 이용해도 빠른 시간에 k번째 소수를 구할 수 있으므로 도전해보기 바란다.

문제 3

소수 구하기(L)

소수(prime number)는 1과 자신을 제외하고는 약수가 없는 수이다. 어떤 수에서 자릿수의 위치를 바꾸었을 때 소수의 여부가 달라질 수 있다.

예를 들어, 23은 소수이지만, 수를 바꾸어 32가 되면 소수가 아니다. 입력되는 정수의 자릿수를 바꾸어서 만들어질 수 있는 소수를 출력하는 프로그램을 작성하시오.

예를 들어, 113의 자릿수를 바꾸면 113, 131, 311을 만들 수 있고 (자신도 포함), 이들 중에서 소수는 113, 131, 311이다.

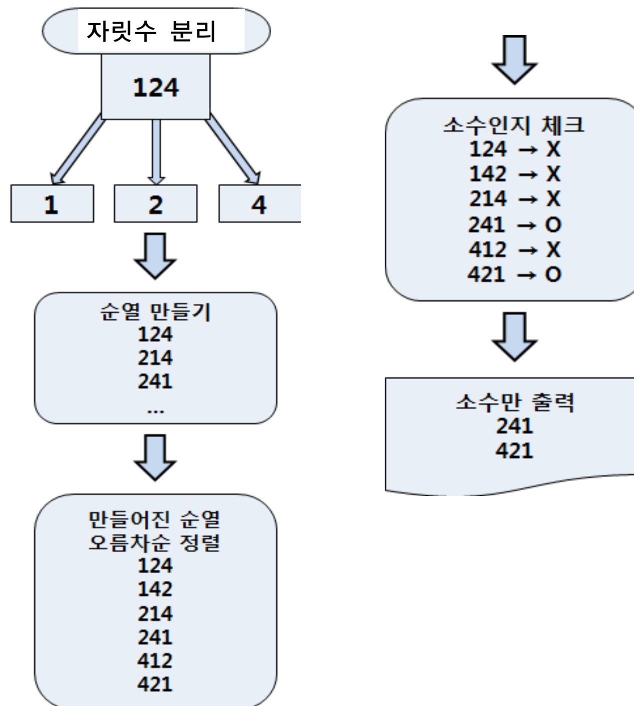
출력

1. n의 자릿수를 바꾸어서 만들어지는 모든 소수를 하나의 공백으로 분리하여 출력한다.
2. 출력되는 수는 크기가 작은 수부터 큰 수의 순으로 출력한다.
3. 같은 수가 중복되어 출력되면 안 된다.
4. 만들어지는 소수가 없으면 0을 출력한다.

입력 예	출력 예
25	0
131	113 131 311
1003	13 31 103 3001
1234	1423 2143 2341 4231

풀이

문제는 얼핏 보면 쉬워 보이지만 실제로 코딩을 해보면 만만치 않은 문제이다. 이 문제를 푸는데 사용되는 알고리즘으로는 숫자 자릿수 분리하기, 순열 만들기, 자료 정렬하기, 소수 확인하기 등을 들 수 있다. 그렇다면 프로그램의 흐름을 설계해 보자. 예를 들어 124라는 숫자가 입력되었을 때를 가정하고 처리방법을 생각해 보자.



이와 같이 풀면 쉽게 풀 수 있을 것이다. 다시 한 번 정리하여 알고리즘을 설계하면 다음과 같다.

1. 입력된 수의 자릿수를 분리하여 배열에 넣는다.
2. 분리된 각 자릿수로 모든 경우의 순열을 만든다.
3. 만들어진 순열들을 오름차순으로 정렬한다.
4. 각각의 순열에 대해 소수인지 확인한다.
5. 소수이고 한 번도 출력하지 않았다면 출력한다.
6. 소수가 하나도 없었으면 -1을 출력한다.

전체 소스 코드는 다음과 같다.

줄 | 코드 | 참고

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  int a[6];
5  int allnum[2000], alli;
6
7  int compare(const void *a, const void *b)
8  {
9     return *(int *)a-*(int *)b;
10 }
11 int chk(int n)
12 {
13   int i;
14   if(n==1)
15    return -1;
16
17   for (i=2; i<n; i++)
18     if(n%i==0) return -1;
19   return 0;
20 }
21 void swap(int *a, int *b)
22 {
23   int temp;
24   temp=*a; *a=*b; *b=temp;
25 }
26 void P(int i, int last)
27 {
28   int num=0, j, k;
29   if(i==last)
30   {
31     for(k=0; k<=last; k++)
32       num+=a[k]*(int)pow((double)10,(double)k);
33     allnum[alli++] = num;
34   }
35   else
36     for(j=i; j<=last; j++)
37     {
38       swap(&a[i], &a[j]);
39       P(i+1, last);
40       swap(&a[i], &a[j]);
41     }
42 }
```

참고:
18: 0: 소수임
21: 자료 스왑
26: 순열 만들기 알고리즘

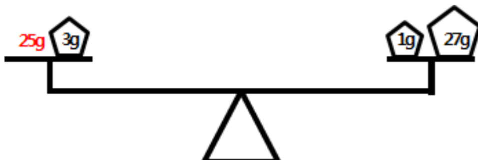
줄	코드	참고
1	`int main()`	5: 배열에 각 자릿수 넣기 8: 배열의 낮은 첨자가 1의 자리수 10: 마지막 자료 인덱스(몇 자리수인지) 13: 순열 만들기 17: 소수이면 25: 소수가 없었으면 -1
2	`{`	
3	`  int n, i, last, prev = -1;`	
4	`  scanf("%d", &n);`	
5	`  for(i=0; i<6; i++)`	
6	`    if(n!=0)`	
7	`    {`	
8	`      a[i]=n%10;`	
9	`      n/=10;`	
10	`      last = i;`	
11	`    }`	
12		
13	`  P(0, last);`	
14	`  qsort(allnum, alli, sizeof(int), compare);`	
15	`  for(i=0; i<alli; i++)`	
16	`  {`	
17	`    if(chk(allnum[i])==0)`	
18	`      if(prev!=allnum[i])`	
19	`      {`	
20	`         printf("%d ", allnum[i]);`	
21	`         prev = allnum[i];`	
22	`      }`	
23	`  }`	
24		
25	`  if(prev==-1)`	
26	`    printf("0");`	
27		
28	`  return 0;`	
29	`}`	

문제 4

저울 추(L)

평형저울을 이용하여 1kg 이하의 물건의 무게를 재려고 한다. 준비되어 있는 추는 1g, 3g, 9g, 27g, 81g, 243g, 729g과 같이 7개의 추뿐이다.

평형저울의 양쪽 접시에 물건과 추를 적절히 놓음으로서 물건의 무게를 잴 수 있는데, 예를 들어, 25g의 물건을 재기 위해서는 다음과 같이 저울에 올려놓으면 된다.



물건의 무게가 입력되었을 때 양쪽의 접시에 어떤 추들을 올려놓아야 평형을 이루는지를 결정하는 프로그램을 작성하시오.

입력

1. 물건의 무게를 나타내는 하나의 정수 n이 입력된다(1≤n≤1,000).
2. n은 물건의 무게가 몇 그램인지를 나타낸다.

출력

1. 저울의 왼쪽 접시와 오른쪽 접시에 올린 추를 0으로 구분하여 출력한다.
2. 각 접시에 올린 추들을 무게가 가벼운 추부터 하나의 공백으로 구분하여 출력한다.
3. 물건의 무게를 왼쪽 접시의 처음에 표시한다.

입력 예	출력 예
25	25 3 0 1 27
40	40 0 1 3 9 27

풀이

이 문제를 보면 생각나는 것이 백트래킹일 것이다. 입력된 수에 각각의 추를 더해가며 모든 경우의 수에 대해서 백트래킹을 하면 구할 수 있을 것이다. 하지만 이번 풀이에서는 백트래킹이 아닌 다른 방법을 제시한다. 문제를 다시 한 번 읽어 보고 생각해 보자. 어떤 특이한 점이 발견되는가?

처음 드는 의문점은 과연 7개의 추를 잘 조절하여 정말 어떤 수와도 균형을 이룰 수 있는가이다. 몇 가지 테스트를 손으로 해보았다. 정말 신기하게도 임의의 수와 균형을 이루는 것을 확인할 수 있다.

그러면서 드는 의문이 왜 하필 추의 무게가 1g, 3g, 9g, 27g, 81g, 243g, 729g인가이다. 추의 무게 수열을 보니 이것은 첫 항이 1이고 등비가 3인 등비수열임을 알 수 있었다. 그리고 이것은 어쩌면 3진법을 활용한 것이 아닐까 하는 의심을 해보아야 한다(왜냐하면 이 수들은 3진수의 자릿수 값이기 때문이다. $1=3^0$, $3=3^1$, $9=3^2$, ...).

일단 3진수로 바꿔서 풀이 방법을 생각해 보자. 숫자 5를 예를 들어 보자.

물건		729	243	81	27	9	3	1
5	=						1	2

〈5를 3진수로 변환〉

⑤ 〈바뀐 3진수를 저울 그림으로 도식화〉

그림을 그리고 보니 1g 추 2개가 있는 것이 거슬린다. 왜냐하면 각각의 추는 하나씩 밖에 없기 때문이다. 1g추 2개를 없앨 수 있는 방법을 생각하게 된다. 저울의 왼쪽에 1g을 추가하면 어떻게 되는가? 그럼 오른쪽에도 1g 추를 추가 하면 될 것이다. 그럼 다음 그림과 같게 된다.



그리고 저울의 오른쪽에 1g 추 3개를 3g 추로 바꿔보자. 그럼 1g추는 한 개만 사용한 것으로 되고 추의 균형은 이루고 있다. 그런데 이번에는 3g 추가 2개가 되었다.



이것을 3진수로 나타내면 다음과 같다.

물건+추		729	243	81	27	9	3	1
5 +1	=						2	0

〈5+1을 3진수로 변환〉

점점 감이 오기 시작한다. 여기에 저울의 좌우에 3g 추를 추가 시켜 균형을 맞추어 보자.



앞에서 한 것처럼 오른쪽에 3g 추 3개를 9g 추로 바꾸자. 그러면 저울의 좌측에 3g 추가 추가되고 추들이 한 번씩만 사용되고 균형도 이루게 된다.



물건+추		729	243	81	27	9	3	1
5+1+3	=					1	0	0

〈5+1+3을 3진수로 변환〉

이런 방법으로 알고리즘을 구현하면 이 문제는 쉽게 풀 수 있다. 이번엔 3진수로 변환된 수에서 살펴보자. 앞의 그림들과 3진수로 변환된 것을 비교하면서 보길 바란다.

3진수로 변환된 수에서 최저 자리 값에서부터 탐색하여 2가 있는 것은 그 자리 값을 더하기 하여 0으로 만들어 버리고 저울의 좌측에 그 자리 값을 더하면 된다.

좌우를 계속 균형 있게 맞추어 가며 오른쪽 자리 값에 2인 것만 0으로 다 변환하면 결과는 쉽게 나올 것이다. 결론적으로 자리 값이 2인 것은 저울의 왼쪽에 더해질 것이고, 자리 값이 1인 것은 그대로 남아서 저울의 오른쪽 값이 될 것이다.

전체 소스는 다음과 같다. 앞의 그림을 그대로 보여주기 위해 루프를 돌때마다 계속 3진수로 바꾸고 아랫자리에서부터 2를 찾는 과정을 넣었다.

이 소스가 이해가 된다면 3진수로 일일이 바꾸지 않고 입력된 원 숫자에서 단 한 번 3진수로 바꾸는 과정에서 모든 계산이 이루어지게 코딩할 수 있다. 도전해 보기 바란다.

줄	코드	참고
1	`#include<stdio.h>`	3: 추가 저장된 배열 4: 출력될 양식을 저장 5: 3진수를 저장하는 배열(0번방이 낮은 자리 수) 6: n: 입력된 수, idx: output배열의 인덱스 11: 3진수로 변환하는 함수 14: 3진법의 자릿수가 2인 자리 찾기 17: 2인 자릿수를 찾으면 19: n에 해당 자리 추를 더함으로서 값이 변함 20: 출력배열에 해당 추를 추가 시킴 21: 자릿수에 2가 있었다면 2를 리턴
2		
3	`int w[7]={1, 3, 9, 27, 81, 243, 729};`	
4	`int output[9];`	
5	`int three[7];`	
6	`int i, j, n, idx;`	
7		
8	`void to3(int num)`	
9	`{`	
10	`  for(int i=0; num!=0; i++, num=num/3)`	
11	`    three[i]=num%3;`	
12	`}`	
13		
14	`int chk2(void)`	
15	`{`	
16	`  for(int i=0; i<7; i++)`	
17	`    if(three[i]==2)`	
18	`    {`	
19	`      n=n+w[i];`	
20	`      output[idx++]=w[i];`	
21	`      return 2;`	

줄	코드	참고

```
22      }
23    return 0;
24  }
25
26  int main(void)
27  {
28    scanf("%d", &n);
29    output[idx++]=n;
30
31    do
32      to3(n);
33    while (chk2()==2);
34
35    idx++;
36    for(i=0; i<7; i++)
37      if(three[i]==1)
38        output[idx++]=w[i];
39
40    for(i=0; i<idx; i++)
41      printf("%d ", output[i]);
42    return 0;
43  }
```

참고

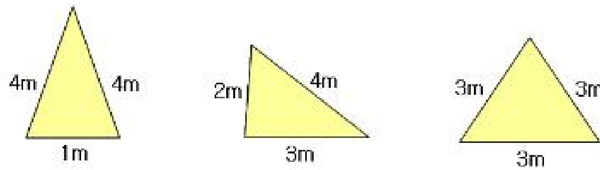
23: 자릿수에 2가 없었다면 0을 리턴
28: 자료 입력
29: 출력의 첫 번째는 입력된 수가 됨
32: 3진법으로 바꾸고
33: 자리 값에 2가 하나라도 있으면 반복
35: 저울의 중심을 나타내기 위해 0을 저장해야 하는데 이미 배열의 초기 값이 0이므로 그냥 인덱스만 더함
37: 자리 값이 1인 것(오른쪽 저울의 추)은
38: output배열에 차례대로 저장

문제 5

삼각화단 만들기(L)

주어진 화단 둘레의 길이를 이용하여 삼각형 모양의 화단을 만들려고 한다. 이 때 만들어진 삼각형 화단 둘레의 길이는 반드시 주어진 화단 둘레의 길이와 같아야 한다. 또한, 화단 둘레의 길이와 각 변의 길이는 자연수이다. 예를 들어, 만들고자 하는 화단 둘레의 길이가 9m라고 하면,

- 한 변의 길이가 1m, 두 변의 길이가 4m인 화단,
- 한 변의 길이가 2m, 다른 변의 길이가 3m, 나머지 변의 길이가 4m인 화단,
- 세 변의 길이가 모두 3m인 3가지 경우의 화단을 만들 수 있다.



화단 둘레의 길이를 입력받아서 만들 수 있는 서로 다른 화단의 수를 구하는 프로그램을 작성하시오.

입력

화단의 길이 n이 주어진다(단, 1 <= n <= 50,000).

출력

출력내용은 입력받은 n으로 만들 수 있는 서로 다른 화단의 수를 출력한다.

입력 예	출력 예
9	3

출처: 한국정보올림피아드(2002 전국본선 초등부)

풀이

이 문제는 앞에서 풀었던 문제와 입력제한 크기 이외에는 같은 문제이다. 앞의 문제에서는 n의 최댓값이 100이었지만 이 문제에서는 50,000으로 증가했다. 따라서 전체탐색으로는 풀 수 없는 문제이다. 다음 소스는 전체탐색으로 해결했던 소스이다.

```
#include <stdio.h>

int n;

int solve()
{
  int cnt = 0;
  scanf("%d", &n);
  for(int a=1; a<=n; a++)
    for(int b=a; b<=n; b++)
      for(int c=b; c<=n; c++)
        if(a+b+c==n && a+b>c)
          cnt++;
  return cnt;
}

int main()
{
  printf("%d\n", solve());
}
```

위 알고리즘에서 순차적으로 a, b, c의 값을 정하면서 탐색을 한다. 하지만 a, b까지만 정하면 c는 $a+b+c=n$이라는 공식으로 쉽게 구할 수 있다. 따라서 $O(n^3)$인 알고리즘을 이 방법으로 $O(n^2)$으로 만들 수 있다. 이를 구현한 소스 코드는 다음과 같다.

줄	코드	참고
1	`#include<stdio.h>`	
2		
3	`int main(void)`	
4	`{`	
5	`  int n, a, b, c, count=0;`	
6	`  scanf("%d", &n);`	
7	`  for(a=1; a<=n; a++)`	
8	`    for(b=1; b<=n; b++)`	
9	`    {`	
10	`      c=n-(a+b);`	
11	`      if(a+b>c && (a<=b && b<=c)) count++;`	
12	`    }`	
13	`  printf("%d\n", count);`	
14	`  return 0;`	
15	`}`	

위 알고리즘의 10행 덕분에 반복문 하나를 줄일 수 있다. 따라서 탐색공간이 많이 줄어 들었다.

여기서 조금 더 탐색 공간을 배제할 수 있는 아이디어를 생각해보자. 삼각형의 둘레의 길이를 n 각 변의 길이를 오름차순으로 정렬한 결과를 $a,\ b,\ c$ 라고 할 때, 가장 긴 변의 길이 c 는 다음 부등식을 만족한다.

$$\left[\frac{n}{3}\right] \le c < \frac{n}{2}\,(n\text{은 2의 배수})\ ,\ \left[\frac{n}{3}\right] \le c < \left[\frac{n}{2}\right]\,(\text{그 외})$$

다음으로 가장 짧은 변 a 는 다음 조건을 만족한다.

$$1 \le a \le \left[\frac{n}{3}\right]$$

이 두 조건을 이용하여 추가적으로 탐색공간을 배제할 수 있다. 이를 적용하여 작성한 소스코드는 다음과 같다.

```c
#include<stdio.h>

int main(void)
{
  int n, a, b, c, count=0;
  scanf("%d", &n);
  for(c=n/3; c<=n/2; c++)
    for(a=1; a<=n/3; a++)
    {
      b=n-(a+c);
      if(a+b>c && (a<=b && b<=c)) count++;
    }
  printf("%d\n", count);
  return 0;
}
```

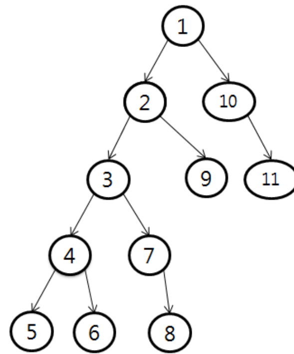
나. 경험적 배제를 이용한 설계

경험적 배제는 일단 앞 단원에서 학습한 전체탐색법을 기본으로 한 알고리즘 설계 방법이다. 처음 시작은 전체탐색과 마찬가지로 해가 될 수 있는 모든 공간을 탐색해 나간다. 차이점은 특정 조건을 두고, 이 조건을 기준으로 다음 상태를 계속 탐색할지의 여부를 결정한다.

여기서의 특정 조건이란, 더 이상 탐색하더라도 해를 구할 수 없음을 판단할 수 있는 조건을 말한다. 이 조건의 설정은 알고리즘이 시작될 때는 정할 수 없고, 탐색을 진행하는 중에 조건을 설정하고, 탐색한 영역이 넓어질수록 상황에 따라 조건이 갱신된다. 따라서 탐색한 정보, 즉 경험한 정보를 이용해서 배제할 조건을 정하기 때문에 경험적 배제라고 한다.

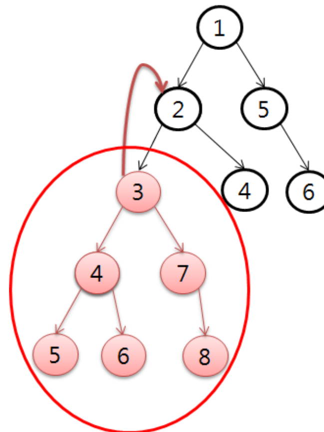
경험적 배제는 일반적으로 가지치기(branch & bound)라고 한다. 이는 마치 탐색구조를 나무로 비유하고, 탐색하지 않는 분기에 대해서 자르는 것이 마치 나무를 관리할 때 가지를 쳐 내는 것과 유사하여 붙여진 이름이다.

다음과 같은 탐색구조가 가지는 문제가 있다.



어떤 문제의 탐색구조

위 구조에서 각 번호는 탐색할 순서이다. 만약 2번에서 3번으로 진행하려고 할 때, 3번 정점이 알고리즘에서 설정한 조건을 만족한다면 3번 정점 이하의 모든 정점들을 더 이상 탐색할 필요가 없으며, 바로 9번으로 진행할 수 있다.



탐색 공간의 배제

위 그림은 더 이상 필요 없음을 판단한 영역을 배제하고 탐색한 결과를 나타낸다. 이는 결과적으로 11회 탐색해야 할 문제를 6회의 탐색으로 동일한 결과를 얻을 수 있기 때문에 알고리즘의 효율을 향상시킬 수 있다.

일반적으로 더 이상 탐색할 정점이 없어서 되돌아오는 것을 백트랙이라고 한다. 하지만 위의 예와 같이 3번 정점에서 되돌아 온 흐름은 백트랙과는 다르다. 이렇듯 어떤 조건에

의해서 더 탐색할 공간이 있음에도 불구하고 돌아오는 흐름을 바운딩(bounding) 혹은 커팅(cutting)라고 한다.

바운딩은 우리가 공을 벽에 던지면 튕겨 나오는 상태를 말한다. 마치 3번 정점이 벽과 같이서 흐름이 튕기는 것처럼 느껴지기 때문에 바운딩이라는 용어를 쓴다. 이 용어를 이해하면 branch & bound라는 이름의 의미를 알 수 있다.

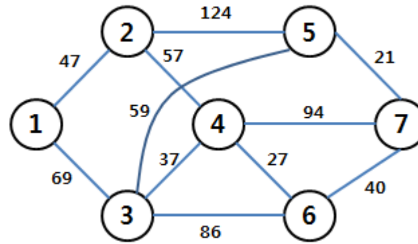
경험적 배제 기법의 핵심은 더 이상 탐색할 필요가 없는 지점을 판단하는 기준을 정하는 것이다. 이 판단의 근거는 일반적으로 탐색 중에 얻을 수 있는 정보를 활용하는 경우가 대부분이다. 앞에서 다루었던 전체탐색법의 예제들 중 분기한정으로 효율을 향상시킬 수 있는 예제를 통하여 조건을 설정하는 방법을 익혀보자.

문제 1

연구활동 가는 길(L)

정올이는 GSHS에서 연구활동 교수님을 뵈러 A대학교를 가려고 한다. 출발점과 도착점을 포함하여 경유하는 지역 n개, 한 지역에서 다른 지역으로 가는 방법이 총 m개이며 GSHS는 지역 1이고 S대학교는 지역 n이라고 할 때 대학까지 최소 비용을 구하시오.

다음 그래프는 예를 보여준다.



최소 비용이 드는 경로 : 1→3→5→7, 최소 비용 : 69+59+21=149

입력

첫 번째 줄에는 저점의 수 n과 간선의 수 m이 공백으로 구분되어 입력된다. 다음 줄부터 m줄에 걸쳐서 두 정점의 번호와 가중치가 입력된다(자기 간선, 멀티 간선이 있을 수 있다.).

출력

대학까지 가는데 드는 최소 비용을 출력한다. 만약 갈 수 없다면 "-1"을 출력.

입력 예	출력 예
7 11 1 2 47 1 3 69 2 4 57 2 5 124 3 4 37 3 5 59 3 6 86 4 6 27 4 7 94 5 7 21 6 7 40	 149

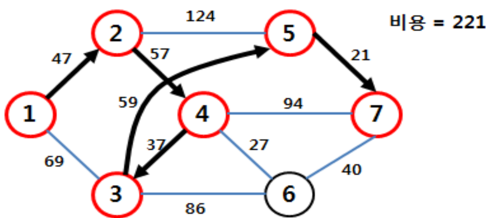
풀이

이 문제는 앞에서 전체탐색법으로 이미 해결했던 문제이다. 하지만 여기서 탐색을 배제할 조건을 설정하여 탐색영역을 줄여보자.

먼저 탐색배제 조건을 설정해야 한다. 이 문제에서는 전체의 최소 이동거리를 구하는 것이므로 탐색 중 임의의 한 경로를 찾았을 때마다 새로운 거리를 구할 수 있으므로 탐색 중 다음과 같은 배제 조건을 설정할 수 있다.

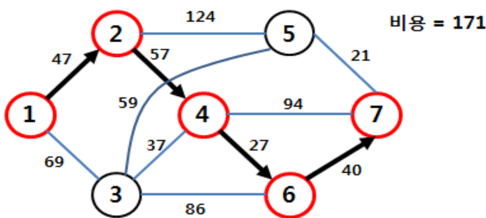
현재 탐색한 거리 > 지금까지 구한최소 경로의 거리

위 조건을 만족할 경우, 더 이상 탐색하지 않더라도 해를 구하는 데 전혀 문제가 없음을 알 수 있다. 이 조건을 적용하여 탐색하는 과정의 일부를 살펴보자.



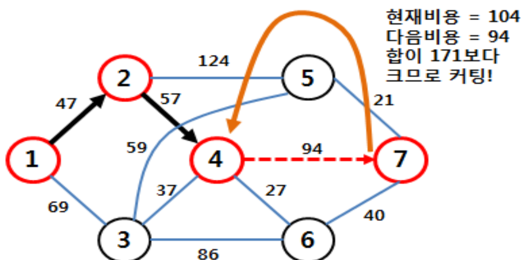
처음으로 찾게 되는 경로, 이 경로의 길이는 221이다.

현재까지 구한 최소 이동거리 = 221



다음으로 구한 경로는 171이 된다. 이 해는 지금까지의 해보다 221보다 더 좋은 경로이므로 갱신

현재까지 구한 최소 이동거리 = 171



다음 경로로 진행하는 도중에 현재까지 최소인 171보다 커지게 되므로 커팅!! 따라서 탐색영역이 배제되고 효율은 높아진다. 이러한 과정으로 마지막까지 진행.

위와 같은 단계를 거치면서 진행하게 되면 해는 점점 더 좋아지고 커팅의 효율은 더 높아진다. 위의 방법으로 작성한 소스코드는 다음과 같다.

코드	참고

```c
#include <stdio.h>
int n, m, G[11][11], sol=0x7fffffff, chk[11];

void solve(int V, int W)
{
  if(W>sol) return;
  if(V==n)
  {
    if(W<sol) sol=W;
      return;
  }
  for(int i=1; i<=n; i++)
    if(!chk[i] && G[V][i])
    {
      chk[i]=1;
      solve(i, W+G[V][i]);
      chk[i]=0;
    }
}

int main(void)
{
  scanf("%d %d", &n, &m);
  for(int i=0; i<m; i++)
  {
    int s, e, w;
    scanf("%d %d %d", &s, &e, &w);
    G[s][e]=G[e][s]=w;
  }
  solve(1, 0);
  printf("%d\n", sol==0x7fffffff ? -1:sol);
  return 0;
}
```

성능 검증을 하기 위하여 counter이라는 변수를 이용하여 해를 구하기까지 몇 개의 상태를 탐색하는지 카운팅하는 프로그램을 작성하고, 3번의 임의의 입력데이터를 이용하여 테스트를 해 보자.

검증하는 프로그램과 검증 데이터 셋은 다음과 같다.

```
#include <stdio.h>
int n, m, G[11][11], sol=0x7fffffff, chk[11];
int counter;

void solve(int V, int W)
{
  if(W>sol) return;
  counter++;
  if(V==n)
  {
    if(W<ol) sol=W;
    return;
  }
  for(int i=1; i<=n; i++)
    if(!chk[i] && G[V][i])
    {
      chk[i]=1;
      solve(i, W+G[V][i]);
      chk[i] = 0;
    }
}

int main(void)
{
  scanf("%d %d", &n, &m);
  for(int i=0; i<m; i++)
  {
    int s, e, w;
    scanf("%d %d %d", &s, &e, &w);
    G[s][e]=G[e][s]=w;
  }
  solve(1, 0);

  printf("%d\n", sol==0x7fffffff ? -1:sol);
  printf("[탐색한 정점 수 %d개]\n", counter);
  return 0;
}
```

다음은 테스트 한 입력데이터 3개이다.

입력 예 1	입력 예 2	입력 예 3
5 7 1 2 2 1 3 10 1 4 7 2 5 4 2 3 6 4 5 3 3 5 4	5 8 1 2 2 1 3 1 1 4 3 2 5 2 2 3 1 4 5 3 3 5 2 1 5 14	7 11 1 2 47 1 3 69 2 4 57 2 5 124 3 4 37 3 5 59 3 6 86 4 6 27 4 7 94 5 7 21 6 7 40

위 3개의 데이터에 대한 결과이다.

알고리즘	데이터1	데이터2	데이터3
전체탐색법	36	42	73
탐색배제1	10	19	25

다음은 배제된 공간의 비율을 보여준다.

알고리즘	데이터1	데이터2	데이터3
탐색배제 1	72.3%	54.77%	65.76%

위 표에서 알 수 있듯이 탐색한 정점의 수가 많이 줄어든 것을 알 수 있다. 대략적으로 원래 방법보다는 2배 이상 빨라졌음을 알 수 있다. 이는 데이터의 특성에 따라 달라질 수 있으니 참고하기 바란다.

이와 같은 알고리즘의 효율은 처음에 구한 해가 얼마나 질이 좋은 해인가에 따라 결정된다. 그렇다면 초반에 질이 좋은 해를 어떻게 구할 수 있을까? 앞에서 다룬 내용 중에 단순 탐욕법이라는 것이 있었다. 이는 현재 상태에서 수학적인 검증 없이 가장 유리한 상태만을 탐색하는 방법이다. 이 방법이 최적해를 구할 수는 없지만 비교적 질이 좋은 해를 구할 수 있다는 사실을 다루었다.

따라서 단순 탐욕법을 이용하여 처음에 하나의 해를 구한다. 일반적으로 이 해가 품질이 좋을 확률이 높으므로 이 해를 처음해로 탐색배제 조건의 기준이 된다. 그리고 위 알고리즘을 실행하면 평균적인 효율이 향상될 가능성이 크다.

단순 탐욕법으로 처음 해를 구하는 소스코드를 추가한 알고리즘은 다음과 같다.

줄	코드	참고
1–33	(아래 코드)	25: bounding (cutting)

```
1  #include <stdio.h>
2  int n, m, G[1001][1001], sol, chk[1001], greedy_chk[1001];
3
4  void greedy_ans(int V)
5  {
6    int W=0, t;
7    greedy_chk[V]=1;
8    while(V!=n)
9    {
10     int min=0x7fffffff;
11     for(int i=1; i<=n; i++)
12       if(!greedy_chk[i] && G[V][i] && G[V][i]<min)
13       {
14         greedy_chk[i]=1;
15         min=G[V][i];
16         t=i;
17       }
18     sol+=G[V][t];
19     V=t;
20   }
21 }
22
23 void solve(int V, int W)
24 {
25   if(W>sol) return;
26   if(V==n)
27   {
28     if(W<sol) sol=W;
29       return;
30   }
31   for(int i=1; i<=n; i++)
32     if(!chk[i] && G[V][i])
33     {
```

```
      chk[i]=1;
      solve(i, W+G[V][i]);
      chk[i]=0;
    }
}

int main(void)
{
  scanf("%d %d", &n, &m);
  for(int i=0; i<m; i++)
  {
    int s, e, w;
    scanf("%d %d %d", &s, &e, &w);
    G[s][e]=G[e][s]=w;
  }
  greedy_ans(1);
  solve(1, 0);
  printf("%d\n", sol==0x7fffffff ? -1:sol);
  return 0;
}
```

위 알고리즘에서 greedy_ans 함수가 처음 해를 단순 탐욕법으로 구하고 있는 과정을 나타낸다. 단순 탐욕법으로 구한 해는 정답이 아닐 가능성이 크지만, 해의 품질이 좋기 때문에 커팅의 조건으로 적합하다. 단순 탐욕법은 이와 같이 다양한 응용이 가능하다.

마지막으로 전체의 효율을 비교한 결과는 다음과 같다.

알고리즘	데이터1	데이터2	데이터3
전체탐색법	36	42	73
탐색배제 1	10	19	25
탐색배제 2	4	8	16

다음은 배제된 공간의 비율을 보여준다.

알고리즘	데이터1	데이터2	데이터3
탐색배제 1	72.3%	54.77%	65.76%
탐색배제 2	89.9%	81.96%	78.09%

이와 같이 원래 알고리즘 보다 4배 이상 효율이 향상되었음을 알 수 있다. 이와 같이 탐색을 배제하는 방법은 정해진 것이 없고, 여기에서 소개한 방법은 가장 기본적인 배제 방법이다.

여기서 소개한 방법 이외에도 다양한 조건을 설정할 수 있으므로 공간을 배제할 방법을 스스로 설정하여 조건을 추가하면 효율이 좋아질 수 있으므로, 항상 창의적인 사고력을 기를 수 있도록 연습하자.

문제 2

minimum sum(M)

n*n개의 수가 주어진다. (1<=n<=10)

이때 겹치지 않는 각 열과 각 행에서 수를 하나씩 뽑는다.
(즉, 총 n개의 수를 뽑을 것이다. 그리고 각 수는 100 이하의 값이다.)

이 n개의 수의 합을 구할 때 최솟값을 구하시오.

입력

첫 줄에 n이 입력된다. 다음 줄부터 n+1줄까지 n개씩의 정수가 입력된다.

출력

구한 최소 합을 출력한다.

입력 예	출력 예
3 1 2 5 2 4 3 5 4 3	7

풀이

이 문제는 앞에서 다루었던 문제이다. 전체탐색으로 간단하게 해결했지만 여기서는 탐색 영역을 배제하여 알고리즘의 효율을 향상시켜보자. 이 문제의 목적은 각 원소들의 합을 최소화하는 것이다. 따라서 현재까지 탐색하고 있는 합이 지금까지 구해 둔 최솟값 이상이 되면 더 이상 탐색할 필요가 없다.

따라서 다음과 같은 배제 조건을 설정할 수 있다.

현재까지의 합 > 지금까지 최소 합

탐색 중 이 조건을 만족하면 탐색을 배제하여 효율을 높일 수 있다. 이 조건을 추가한 소스코드는 다음과 같다.

줄	코드	참고

```
1  #include <stdio.h>
2  int m[11][11];
3  int col_check[11];
4  int n, min_sol=0x7fffffff;
5  
6  void input(void)
7  {
8    scanf("%d", &n);
9    for(int i=0; i<n; i++ )
10     for(int j=0; j<n; j++)
11       scanf("%d", &m[i][j]);
12 }
13 
14 void solve(int row, int score)
15 {
16   if(score>min_sol) return;
17   if(row==n)
18   {
19     if(score<min_sol)
20       min_sol=score;
21     return;
22   }
23   for(int i=0; i<n; i++)
24   {
```

참고: 16: bounding (cutting)

```
    if(col_check[i]==0)
    {
      col_check[i]=1;
      solve(row+1, score+m[row][i]);
      col_check[i]=0;
    }
  }
  return;
}
```

위 알고리즘에서 단순 탐욕법으로 해를 구하는 구문을 추가하여 효율을 비교해보자. 단순 탐욕법으로 처음 해를 구하는 과정은 다음과 같다.

1. 1행에서 가장 작은 수를 택하고 다음 행으로 진행한다.
2. 다음 행에서 아직까지 선택되지 않은 열 중 가장 작은 수를 택하고 다음 행으로 진행한다.
3. 아직 마지막 행을 마치지 않았으면 2단계로 간다.
4. 지금까지 선택한 수들의 합을 처음 해로 한다.

이 과정을 소스코드로 작성하면 다음과 같다.

```
#include <stdio.h>
int m[11][11];
int col_check[11], greedy_chk[11];
int n, min_sol=0;

void input(void)
{
  scanf("%d", &n);
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      scanf("%d", &m[i][j]);
}

void greedy_ans(int row)
{
  for(int i=row; i<n; i++)
  {
    int min=0x7fffffff, k;
    for(int j=0; j<n; j++)
```

줄	코드	참고

```
20      if(!greedy_chk[j] && min>m[i][j])
21      {
22        min=m[i][j];
23        k=j;
24      }
25    min_sol+=min;
26    greedy_chk[k]=1;
27  }
28 }
29
30 int main()
31 {
32   input();
33   greedy_ans(0);
34   solve(0, 0);
35   printf("%d", min_sol);
36   return 0;
37 }
```

강조된 부분이 바뀌거나 추가된 부분이다. 완전탐색법으로 탐색했을 때 탐색한 정점의 수와 2가지 배제 방법을 각각 적용했을 때의 효율을 비교해보자. 비교하기 위하여 다음 코드를 추가한다.

줄	코드	참고
		9: bounding (cutting)

```
1  #include <stdio.h>
2  int m[11][11];
3  int col_check[11], greedy_chk[11];
4  int n, min_sol=987654321;
5  int counter;
6
7  void solve(int row, int score)
8  {
9    if(score>min_sol) return;
10   counter++;
11   if(row==n)
12   {
13     if(score<min_sol)
14       min_sol=score;
15     return;
16   }
```

```
  for(int i=0; i<n; i++)
  {
    if(col_check[i]==0)
    {
      col_check[i]=1;
      solve(row+1, score+m[row][i]);
      col_check[i] = 0;
    }
  }
  return;
}

int main()
{
  input();
  greedy_ans(0);
  solve(0, 0);
  printf("%d\n", min_sol);
  printf("탐색한 정점의 수 : %d\n", counter);
  return 0;
}
```

위 코드를 추가하면 탐색한 정점의 수를 마지막에 출력할 수 있다. 탐색한 정점의 수는 탐색영역의 크기라고 할 수 있으므로 효율이 얼마나 향상되었는지 비교할 수 있다. 다음 주어진 3개의 데이터를 이용해서 전체탐색법, 최솟값 규칙만 적용한 탐색배제 1, 단순 탐욕법까지 추가한 탐색배제 2의 효율을 비교해보자.

입력 예 1	입력 예 2	입력 예 3
3 12 76 2 52 77 37 13 67 16	5 93 61 92 56 94 18 32 17 10 64 20 98 85 32 82 1 45 66 77 78 52 11 94 26 57	7 88 51 24 88 94 50 60 14 55 1 23 12 84 91 26 44 81 97 33 82 30 3 71 12 99 16 92 48 87 5 14 93 28 92 56 4 14 92 96 48 41 77 94 32 43 16 1 52 51

위 3개의 데이터에 대한 결과이다.

알고리즘	데이터1	데이터2	데이터3
전체탐색법	16	326	13700
탐색배제1	10	97	486
탐색배제2	10	79	330

다음은 배제된 공간의 비율을 보여준다.

알고리즘	데이터1	데이터2	데이터3
탐색배제 1	37.5%	70.25%	99.06%
탐색배제 2	37.5%	75.77%	99.08%

위 결과에서 알 수 있듯이 탐색을 배제한 효과는 데이터의 수가 많아질수록 커진다. 따라서 다양한 아이디어로 적절히 잘 배제하면 어려운 문제도 효율적으로 해결할 수 있는 방법이 될 수 있으므로 다양한 아이디어로 배제에 도전하자.

문제 3

거스름 돈(M)

여러분은 실력을 인정받아 전 세계적으로 사용할 수 있는 자동판매기용 프로그램의 개발을 의뢰받았다. 거스름돈에 사용될 동전의 수를 최소화하는 것이다.

입력으로 거슬러 줘야할 돈의 액수와 그 나라에서 이용하는 동전의 가짓수 그리고 동전의 종류가 들어오면 여러 가지 방법들 중 가장 적은 동전의 수를 구하는 프로그램을 작성하시오.

입력

첫 번째 줄에는 거슬러 줘야할 돈의 액수 m이 입력된다.
(10 <= m <= 10,000)
다음 줄에는 그 나라에서 사용되는 동전의 종류의 수 n이 입력된다.
(1 <= n <= 10)
마지막 줄에는 동전의 수만큼의 동전 액수가 오름차순으로 입력된다.
(10 <= 액수 <= m)

출력

최소의 동전의 수를 출력한다.

입력 예	출력 예
730 5 10 50 100 500 1250	6

풀이

이 문제도 전체탐색으로 다루었던 문제이다. 이번 문제도 마찬가지로 최소 동전의 수를 구하는 문제이다. 따라서 앞서 다루었던 문제들과 마찬가지로 간단히 지금까지 구한 최솟값보다 큰 값에 대해서는 배제할 수 있다. 따라서 다음과 같은 배제 조건을 설정할 수 있다.

> 현재까지의 사용한동전의수 > 지금까지 구해둔 최소 동전의 수

위 조건을 추가한 소스코드는 다음과 같다.

```
#include<stdio.h>

int m, n, coin[10], ans=987654321;

void solve(int mon, int d)
{
  if(d>ans) return;
  if(mon>m) return;
  if(mon==m)
  {
    if(d<ans) ans=d;
    return;
  }
  for(int i=0; i<n; i++)
    solve(mon+coin[i], d+1);
}

int main()
{
  scanf("%d%d", &m, &n);
  for(int i=0; i<n; i++)
    scanf("%d", coin+i);
  solve(0, 0);
  printf("%d\n", ans);
  return 0;
}
```

위 알고리즘에서 단순 탐욕법으로 첫 번째 해를 구하는 구문을 추가하여 조금 더 효율을 높일 수 있다. 단순 탐욕법으로 처음 해를 구하는 과정은 다음과 같다.

1. 현재 금액에서 지불할 수 있는 가장 큰 동전으로 가능한 만큼 지불한다.
2. 만약 아직 돈이 남았으면, 다시 1단계로 간다.
3. 지금까지 지불한 동전의 개수를 처음 해로 설정한다.

이 과정을 소스코드로 작성하면 다음과 같다.

```
#include<stdio.h>

int m, n, coin[10], ans=987654321;

void greedy_ans(int mon)
{
  ans=0;
  while(mon)
  {
    for(int i=n-1; i>=0; i--)
    {
      ans+=mon/coin[i];
      mon%=coin[i];
    }
  }
}

void solve(int mon, int d)
{
  if(d>ans) return;
  if(mon>m) return;
  if(mon==m)
  {
    if(d<ans) ans=d;
    return;
  }
  for(int i=0; i<n; i++)
    solve(mon+coin[i], d+1);
}

int main()
{
  scanf("%d%d", &m, &n);
  for(int i=0; i<n; i++)
```

```
    scanf("%d", coin+i);
  greedy_ans(m);
  solve(0, 0);
  printf("%d\n", ans);
  return 0;
}
```

전체탐색과 배제 조건을 하나 적용한 탐색배제1, 단순 탐욕법을 적용한 탐색배제2의 효율을 비교해 보자.

```
#include<stdio.h>

int m, n, coin[10], ans=987654321;
int counter;

void greedy_ans(int mon)
{
  ans=0;
  while(mon)
  {
    for(int i=n-1; i>=0; i--)
    {
      ans+=mon/coin[i];
      mon %= coin[i];
    }
  }
}

void solve(int mon, int d)
{
  if(d>ans) return;
  if(mon>m) return;
  counter++;
  if(mon==m)
  {
    if(d<ans) ans=d;
    return;
  }
  for(int i=0; i<n; i++)
```

```
    solve(mon+coin[i], d+1);
}

int main()
{
  scanf("%d%d", &m, &n);
  for(int i=0; i<n; i++)
    scanf("%d", coin+i);
  greedy_ans(m);
  solve(0, 0);
  printf("%d\n", ans);
  printf("탐색한 정점의 수 %d\n", counter);
  return 0;
}
```

위 소스를 적용하여 다음 3가지 데이터로 측정한다.

입력 예 1	입력 예 2	입력 예 3
180 4 10 50 100 500	380 7 10 50 100 500 1000 5000 10000	570 7 10 50 100 500 1000 5000 10000

위 3개의 데이터에 대한 결과이다.

알고리즘	데이터1	데이터2	데이터3
전체탐색법	409	185,398	62,005,020
탐색배제1	212	7,121	52,800
탐색배제2	111	2,193	151

다음은 배제된 공간의 비율을 보여준다.

알고리즘	데이터1	데이터2	데이터3
탐색배제 1	48.16%	96.15%	99.94%
탐색배제 2	72.86%	98.81%	99.99%

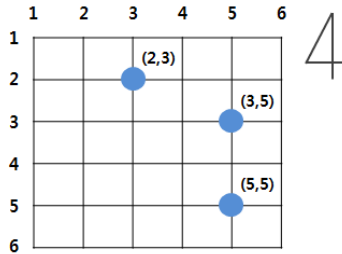
문제 4

경찰차(M)

어떤 도시의 중심가는 n개의 동서방향 도로와 n개의 남북방향 도로로 구성되어 있다.

모든 도로에 는 도로 번호가 있으며 남북방향 도로는 왼쪽부터 1에서 시작하여 n까지 번호가 할당되어 있고 동서방향 도로는 위부터 1에서 시작하여 n까지 번호가 할당되어 있다. 또한 동서방향 도로 사이의 거리와 남북방향 도로 사이의 거리는 모두 1이다.

동서방향 도로와 남북방향 도로가 교차하는 교차로의 위치는 두 도로의 번호의 쌍인 (동서방향 도로 번호, 남북방향 도로 번호)로 나타낸다. n이 6인 경우의 예를 들면 다음과 같다.



이 도시에는 두 대의 경찰차가 있으며 두 차를 경찰차1과 경찰차2로 부른다. 처음에는 항상 경찰차1은 (1, 1)의 위치에 있고 경찰차2는 (n, n)의 위치에 있다.

경찰 본부에서는 처리할 사건이 있으면 그 사건이 발생된 위치를 두 대의 경찰차 중 하나에 알려 주고, 연락 받은 경찰차는 그 위치로 가장 빠른 길을 통해 이동하여 사건을 처리한다(하나의 사건은 한 대의 경찰차가 처리한다.).

그리고 사건을 처리한 경찰차는 경찰 본부로부터 다음 연락이 올 때까지 처리한 사건이 발생한 위치에서 기다린다. 경찰 본부에서는 사건이 발생한 순서대로 두 대의 경찰차에 맡기려고 한다.

처리해야 될 사건들은 항상 교차로에서 발생하며 경찰 본부에서는 이러한 사건들을 나누어 두 대의 경찰차에 맡기되, 두 대의 경찰차들이 이동하는 거리의 합을 최소화하도록 사건을 맡기려고 한다.

경찰차(M) (계속)

예를 들어 앞의 그림처럼 n=6인 경우, 처리해야 하는 사건들이 3개 있고 그 사건들이 발생된 위치를 순서대로 (3, 5), (5, 5), (2, 3)이라고 하자.

(3, 5)의 사건을 경찰차2에 맡기고 (5, 5)의 사건도 경찰차2에 맡기며, (2, 3)의 사건을 경찰차1에 맡기면 두 차가 이동한 거리의 합은 4 + 2 + 3 = 9가 되고, 더 이상 줄일 수는 없다.

처리해야 할 사건들이 순서대로 주어질 때, 두 대의 경찰차가 이동하는 거리의 합을 최소화하는 프로그램을 작성하시오.

입력

입력 파일의 첫째 줄에는 동서방향 도로의 개수를 나타내는 정수 n(3≤n≤1,000)이 주어진다.

둘째 줄에는 처리해야 하는 사건의 개수를 나타내는 정수 w(1≤w≤15)가 주어진다.

셋째 줄부터 (w+2)번째 줄까지 사건이 발생된 위치가 한 줄에 하나씩 주어진다. 경찰차들은 이 사건들을 주어진 순서대로 처리해야 한다.

각 위치는 동서방향 도로 번호를 나타내는 정수와 남북방향 도로 번호를 나타내는 정수로 주어지며 두 정수 사이에는 빈 칸이 하나 있다. 두 사건이 발생한 위치가 같을 수 있다.

출력

첫째 줄에 두 경찰차가 이동한 총 거리를 출력한다.

입력 예	출력 예
6 3 3 5 5 5 2 3	9

출처: 한국정보올림피아드(2003 전국본선 중등부)

풀이

경찰차 문제 또한 최소거리를 구하는 문제이므로, 탐색배제 조건은 앞의 문제들과 마찬가지로 쉽게 설정할 수 있다. 다음과 같은 조건을 설정하자.

현재까지의 이동 거리 > 현재까지의 최소 거리

이 조건을 적용한 소스코드는 다음과 같다.

줄 | 코드 | 참고

```
1  #include <stdio.h>
2
3  int E[1010][2], n, m, ans=987654321;
4
5  int min(int a, int b)
6  {
7    return a>b?b:a;
8  }
9  int abs(int a)
10 {
11   return a>0?a:-a;
12 }
13 int dis(int a, int b)
14 {
15   return abs(E[a][0]-E[b][0])+ abs(E[a][1]-E[b][1]);
16 }
17
18 void solve(int a, int b, int d)
19 {
20   int next=(a>b ? a:b)+1;
21   if(d>ans) return;
22   if(next>m+2)
23   {
24     if(d<ans) ans=d;
25     return;
26   }
27   solve(next, b, d+dis(a, next));
28   solve(a, next, d+dis(b, next));
29 }
30
31 int main()
```

참고: 21 : bounding (cutting)

줄	코드	참고
32	{	
33	scanf("%d %d", &n, &m);	
34	E[0][0]=E[0][1]=1;	
35	E[1][0]=E[1][1]=n;	
36	for(int i=2; i<m+2; i++)	
37	scanf("%d%d",&E[i][0], &E[i][1]);	
38	solve(0, 1, 0);	
39	printf("%d", ans);	
40	return 0;	
41	}	

이 문제도 배제 조건의 효율을 높이기 위해서 처음의 해를 단순 탐욕법으로 구할 수 있다. 방법은 다음과 같이 진행한다.

1단계. 사건dl 발생하면 두 대의 경찰차들 중 더 가까운 경찰차가 사건을 처리한다.
2단계. 만약 아직 남은 사건이 있다면 1단계로 간다.
3단계. 지금까지 이동한 거리의 합을 처음 해로 설정한다.

이 과정을 소스코드로 작성하면 다음과 같다.

줄	코드	참고
1	#include <stdio.h>	33 : bounding (cutting)
2		
3	int E[1010][2], n, m, ans=987654321;	
4		
5	int min(int a, int b)	
6	{	
7	return a>b?b:a;	
8	}	
9	int abs(int a)	
10	{	
11	return a>0?a:-a;	
12	}	
13	int dis(int a, int b)	
14	{	
15	return abs(E[a][0]-E[b][0])+abs(E[a][1]-E[b][1]);	
16	}	
17		
18	void greedy_ans(int a, int b)	
19	{	

```
  ans=0;
  for(int i=2; i<m+2; i++)
  {
    if(dis(i, a)>dis(i, b))
      ans+=dis(i, b), b=i;
    else
      ans+=dis(i, a), a=i;
  }
}

void solve(int a, int b, int d)
{
  int next=(a>b ? a:b)+1;
  if(d>ans) return;
  if(next>=m+2)
  {
    if(d<ans) ans=d;
    return;
  }
  solve(next, b, d+dis(a, next));
  solve(a, next, d+dis(b, next));
}

int main()
{
  scanf("%d%d", &n, &m);
  E[0][0]=E[0][1]=1;
  E[1][0]=E[1][1]=n;
  for(int i=2; i<m+2; i++)
    scanf("%d%d",&E[i][0], &E[i][1]);
  greedy_ans(0, 1);
  solve(0, 1, 0);
  printf("%d", ans);
  return 0;
}
```

탐색 배제의 효율을 알아보기 위해 counter를 추가한 소스는 다음과 같다.

줄	코드	참고

```
1  #include <stdio.h>
2
3  int E[1010][2], n, m, ans=987654321;
4  int counter;
5
6  int min(int a, int b)
7  {
8    return a>b?b:a;
9  }
10 int abs(int a)
11 {
12   return a>0?a:-a;
13 }
14 int dis(int a, int b)
15 {
16   return abs(E[a][0]-E[b][0]) + abs(E[a][1]-E[b][1]);
17 }
18
19 void greedy_ans(int a, int b)
20 {
21   ans=0;
22   for(int i=2; i<m+2; i++)
23   {
24     if(dis(i, a)>dis(i, b))
25       ans+=dis(i, b), b=i;
26     else
27       ans+=dis(i, a), a=i;
28   }
29 }
30
31 void solve(int a, int b, int d)
32 {
33   int next=(a>b ? a:b)+1;
34   if(d>ans) return;
35     counter++;
36   if(next>=m+2)
37   {
38     if(d<ans) ans=d;
39     return ;
40   }
```

참고: 34 : bounding (cutting)

```
  solve(next, b, d+dis(a, next));
  solve(a, next, d+dis(b, next));
}

int main()
{
  scanf("%d %d", &n, &m);
  E[0][0]=E[0][1]=1;
  E[1][0]=E[1][1]=n;
  for(int i=2; i<m+2; i++)
    scanf("%d%d",&E[i][0], &E[i][1]);
  greedy_ans(0, 1);
  solve(0, 1, 0);
  printf("%d\n", ans);
  printf("탐색한 정점의 수는 %d개입니다.\n", counter);
  return 0;
}
```

3가지 데이터를 준비하고 실행한 결과는 다음과 같다.

입력 예 1	입력 예 2	입력 예 3
11 4 5 2 10 8 6 3 7 10	11 6 2 2 4 7 3 3 6 6 6 2 3 5	11 11 9 3 9 3 3 9 3 9 3 3 3 3 3 3 5 9 5 9 10 3 10 3

위 3개의 데이터에 대한 결과이다.

알고리즘	데이터1	데이터2	데이터3
전체탐색법	31	127	4095
탐색배제1	12	28	137
탐색배제2	16	28	98

다음은 배제된 공간의 비율을 보여준다.

알고리즘	데이터1	데이터2	데이터3
탐색배제 1	61.29%	77.95%	98.49%
탐색배제 2	48.38%	77.95%	98.92%

이번 데이터에서는 재미있는 점이 발견된다. 데이터 1의 경우에는 단순 탐욕법을 활용하지 않은 쪽이 더 효율이 좋다. 단순 탐욕법의 해가 항상 질이 좋은 것은 아닐 수 있으므로 발생할 수 있는 일이다. 하지만 데이터가 많아지면 효율이 좋아지는 것은 변함없다.

문제 5

선물(M)

길동이는 세쌍둥이의 첫째이다. 길순이가 둘째이고, 길삼이가 막내이다. 길동 3남매의 생일을 맞이하여 전국 각지에서 친지들이 보내온 수많은 선물이 도착하였다.

길동이 부모는 이 선물들을 길동이 3남매에게 어떻게 나누어 줄 것인가로 고민하고 있다. 선물의 크고 작음 때문에 발생될 수도 있는 남매간의 다툼을 미연에 방지하고자 길동이 가족은 다음과 같이 나누기로 결정하였다.

(1) 선물의 내용을 미리 보지 않고 부피만을 기준으로 배분한다.
(2) 한 사람이 가지는 선물의 개수는 배분의 기준이 아니다.
(3) 선물이 공평하게 나누어 질 수 있도록 3남매가 가지는 선물들의 부피의 합계 차이가 최소가 되도록 한다.
(4) 선물의 부피가 똑같이 나누어지지 못하는 경우에는 길동-길순-길삼의 순으로 합계 부피가 많도록 배분한다.
(5) 3남매가 가지게 되는 부피가 결정되면, 길삼-길순-길동의 순으로 선물을 선택한다.

우리가 길동 부모의 수고를 덜어주고자 길동이 3남매가 가지게 될 선물의 부피를 계산하고자 한다. 선물 부피에 따른 선물 배분의 세부적인 조건은 다음과 같다.

조건 1: 아래의 d가 최소가 되도록 한다.

d = (길동 선물의 부피 합) – (길삼 선물의 부피 합)

조건 2: 같은 d가 되는 배분 방법이 여럿 존재하는 경우에는 길동의 선물의 부피 합이 적은 방법을 선택한다.

선물(M) (계속)

예를 들어, 선물이 6개이고 그 부피가 다음과 같다면,

6, 4, 4, 4, 6, 9

길동은 부피의 합계가 12, 길순은 12, 길삼은 9를 가지도록 배분하면 조건 1에 따라 12-9=3로 최소가 된다.

(길동 13, 길순 10, 길삼 10으로 배분하는 방법도 13-10=3으로 차이가 3이 되지만, 조건 2에 따라 답이 되지 못한다.)

선물의 부피가 입력되었을 때 3남매에게 나누어줄 선물의 합계 부피를 구하는 프로그램을 작성하시오.

입력

1. 첫 줄에 선물의 개수를 나타내는 정수 n가 입력된다(3≤n≤15).
2. 다음 줄에 선물의 부피를 나타내는 n개의 정수가 공백으로 분리되어 입력된다.
3. 선물의 부피는 0보다 크고 100보다 작다

출력

1. 길동 3남매가 가지게 될 선물의 합계 부피를 출력한다.
2. 길동, 길순, 길삼의 순으로 3개의 정수를 하나의 공백으로 분리하여 출력한다.

입력 예	출력 예
6 6 4 4 4 6 9	12 12 9
3 2 10 1	10 2 1
9 1 1 1 4 6 1 1 1 1	6 6 5

풀이

이번 문제에서는 지금까지의 문제와는 달리 최솟값을 구하는 문제가 아니다. 따라서 탐색을 배제할 수 있는 요소들이 다양하다.

먼저 가장 쉽게 시간을 줄이기 위해서 길동, 길순, 길삼이 가지는 선물의 현재 상태로 더 이상 해의 가능성이 없는 경우를 찾는다. 더 이상 가능성이 없다면 탐색을 중지하여 처리 시간을 줄여보자. 길동, 길순, 길삼의 부피에는 다음과 같은 관계가 성립한다.

- A = 길동의 선물 부피 , B = 길순의 선물 부피 , C = 길삼의 선물 부피
- S = 전체 선물무게의 합 이라 할 때,

$$\frac{S}{3} \geq C,\ \frac{S}{2} \geq B$$

위 조건을 추가하면 탐색공간의 배제 효과가 클 것으로 판단된다. 이 조건을 적용한 소스코드는 다음과 같다.

줄	코드	참고

```
1  #include <stdio.h>
2  #include <algorithm>
3
4  int gift[30], chk[30], DIF=0x7fffffff, A, B, C, n, S;
5
6  int comp(int a, int b)
7  {
8    return a>b;
9  }
10
11 void solve( int no, int a, int b, int c )
12 {
13   if(c>S/3 || b>S/2) return;
14   if(no<n)
15   {
16     solve(no+1, a, b, c+gift[no]);
17     solve(no+1, a, b+gift[no], c);
18     solve(no+1, a+gift[no], b, c);
19   }
20   else if((a >= b && b >= c) && a-c && a-c < DIF)
21   {
```

참고: 13: bounding (cutting)

줄	코드	참고
22	`    DIF=a-c, A=a, B=b, C=c;`	
23	`  }`	
24	`}`	
25		
26	`int main()`	
27	`{`	
28	`  int i;`	
29	`  scanf("%d", &n);`	
30	`  for(i=0; i<n; S+=gift[i++])`	
31	`    scanf("%d", &gift[i]);`	
32	`  std::sort(gift, gift+n, comp);`	
33	`  solve(0, 0, 0, 0);`	
34	`  printf("%d %d %d\n", A, B, C);`	
35	`  return 0;`	
36	`}`	

다음으로 또 다른 배제 방법을 생각해 볼 수 있다. 아래의 명제가 참이면 더 이상 탐색할 필요가 없다.

- 남아있는 선물의 부피 + 길동의 선물의 부피 〈 길순의 선물의 부피
- 남아있는 선물의 부피 + 길순의 선물의 부피 〈 길삼의 선물의 부피

위 명제를 식으로 나타내면

$$(S-(a+b+c)) + a < b$$

$$(S-(a+b+c)) + b < c$$

위 조건을 추가한 소스는 다음과 같다.

줄	코드	참고
1	`#include <stdio.h>`	13: bounding1 (cutting) 15: bounding2 (cutting)
2	`#include <algorithm>`	
3		
4	`int gift[30], chk[30], DIF=0x7fffffff, A, B, C, n, S;`	
5		
6	`int comp(int a, int b)`	
7	`{`	
8	`  return a>b;`	
9	`}`	
10		
11	`void solve( int no, int a, int b, int c )`	

```
{
  if(c>S/3 || b>S/2) return;
  int rest = S-(a+b+c);
  if(a+rest<b || b+rest<c) return;
  if(no<n)
  {
    solve(no+1, a, b, c+gift[no]);
    solve(no+1, a, b+gift[no], c);
    solve(no+1, a+gift[no], b, c);
  }
  else if((a>=b && b>=c) && a-c && a-c<DIF)
  {
    DIF=a-c, A=a, B=b, C=c;
  }
}

int main()
{
  int i;
  scanf("%d", &n);
  for(i=0; i<n; S+=gift[i++] )
    scanf("%d", &gift[i]);
  std::sort(gift, gift+n, comp);
  solve( 0, 0, 0, 0 );
  printf("%d %d %d\n", A, B, C);
  return 0;
}
```

그 외에도 커팅 조건을 더 찾을 수 있다. 지금까지 길동과 길삼이 얻은 선물의 부피 차를 DIF라고 하고, 남은 선물의 양을 rest라고 할 때, 다음 조건을 만족한다면 더 이상 탐색할 필요가 없다.

$$A-(C+rest) > DIF$$

위 조건도 추가하면 다음과 같다.

줄	코드	참고

```
1  #include <stdio.h>
2  #include <algorithm>
3
4  int gift[30], chk[30], DIF=0x7fffffff, A, B, C, n, S;
5
6  int comp(int a, int b)
7  {
8    return a>b;
9  }
10
11 void solve( int no, int a, int b, int c )
12 {
13   if(c>S/3 || b>S/2) return;
14   int rest=S-(a+b+c);
15   if(a+rest<b || b+rest<c) return;
16   if(a-(c+rest)>DIF) return
17   if(no<n)
18   {
19     solve(no+1, a, b, c+gift[no]);
20     solve(no+1, a, b+gift[no], c);
21     solve(no+1, a+gift[no], b, c);
22   }
23   else if((a>=b && b>=c) && a-c && a-c<DIF)
24   {
25     DIF = a-c, A = a, B = b, C = c;
26   }
27 }
28
29 int main()
30 {
31   int i;
32   scanf("%d", &n);
33   for(i=0; i<n; S+=gift[i++])
34     scanf("%d", &gift[i]);
35   std::sort(gift, gift+n, comp);
36   solve(0, 0, 0, 0);
37   printf("%d %d %d\n", A, B, C);
38   return 0;
39 }
```

참고:
13: bounding1 (cutting)
15: bounding2
16: bounding3

다양한 커팅의 조건들을 추가한 상태에서 효율을 측정하기 위해 counter 변수를 설정한 소스코드는 다음과 같다.

줄 | 코드

```
1  #include <stdio.h>
2  #include <algorithm>
3  
4  int gift[30], chk[30], DIF=0x7fffffff, A, B, C, n, S;
5  int counter;
6  
7  int comp(int a, int b)
8  {
9    return a>b;
10 }
11 
12 void solve(int no, int a, int b, int c)
13 {
14   if(c>S/3 || b>S/2) return;
15   int rest=S-(a+b+c);
16   if(a+rest<b || b+rest<c) return;
17   if(a-(c+rest)>DIF) return;
18   counter++;
19   if(no<n)
20   {
21     solve(no+1, a, b, c+gift[no]);
22     solve(no+1, a, b+gift[no], c);
23     solve(no+1, a+gift[no], b, c);
24   }
25   else if((a>=b && b>=c) && a-c && a-c<DIF)
26   {
27     DIF=a-c, A=a, B=b, C=c;
28   }
29 }
30 
31 int main()
32 {
33   int i;
34   scanf("%d", &n);
35   for(i=0; i<n; S+=gift[i++])
36     scanf("%d", &gift[i]);
37   std::sort(gift, gift+n, comp);
```

참고

14: bounding1 (cutting)
16: bounding2
17: bounding3

```
  solve(0, 0, 0, 0);
  printf("%d %d %d\n", A, B, C);
  printf("탐색 중 방문한 노드의 수는 %d입니다.\n", counter);
  return 0;
}
```

테스트 할 데이터 3개를 다음과 같이 준비하고 각 배제 적용 방법의 효율을 비교한 표는 다음과 같다.

입력 예 1	입력 예 2	입력 예 3
7 20 1 4 76 30 68 5	10 78 15 74 82 78 85 3 40 44 53	11 94 12 18 38 79 21 38 42 56 93 94

위 3개의 데이터에 대한 결과이다.

알고리즘	데이터1	데이터2	데이터3
전체탐색법	3,208	88,537	265,720
탐색배제 1	988	34,524	108,378
탐색배제 2	572	16,037	49,483
탐색배제 3	15	698	2,299

다음은 배제된 공간의 비율을 보여준다.

알고리즘	데이터1	데이터2	데이터3
탐색배제 1	69.2%	61.0%	59.21%
탐색배제 2	82.16%	81.88%	81.37%
탐색배제 3	99.53%	99.21%	99.13%

이번 배제 기법은 앞에서 봤던 기법들과 달리 데이터의 크기에 관계없이 일정한 비율을 배제하는 것을 알 수 있다. 특히 세 번째 배제기법의 효율이 매우 좋음을 알 수 있다. 물론 이 이외에도 다양한 조건들을 더 추가할 수도 있으므로 다양한 아이디어를 이용할 수 있도록 연습해보자.